SAS/C® Library Reference, Third Edition, Release 6.00

Volume 1

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS/C® Library Reference, Third Edition, Volume 1, Release 6.00*, Cary, NC: SAS Institute Inc., 1995. 777 pp.

SAS/C® Library Reference, Third Edition, Volulme 1

ISBN 1-55544-666-3

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

1st printing, November 1995

Doc P17, 071995

Contents

Using This Book

SAS/C Library Reference, Third Edition, Volume 1, Release 6.00 provides complete reference documentation for the functions that comprise the SAS/C Library. It is primarily intended for experienced C programmers. It makes no attempt to discuss either programming fundamentals or how to program in C.

SAS Online Samples

Many of the examples used in this book are available through SAS Online Samples.

SAS Online Samples enables you to download the sample programs from many SAS books by using one of three facilities: **Anonymous FTP**, **SASDOC-L**, or the **World Wide Web**.

Anonymous FTP

Anonymous FTP enables you to download ASCII files and binary files (SAS data libraries in transport format). To use anonymous FTP, connect to FTP.SAS.COM. Once connected, enter the following responses as you are prompted:

Name (ftp.sas.com:user-id):
  anonymous
Password:
  <your e-mail address>

Next, change to the publications directory:

>cd pub/publications

For general information about files, download the file **info**:

>get info <*target-filename*>

For a list of available sample programs, download the file **index**:

>get index <*target-filename*>

Once you know the name of the file you want, issue a GET command to download the file. Note: Filenames are case sensitive.

To download...	**issue this command...**
compressed ASCII file	**>get *filename*.Z <*target-filename*>**
ASCII file	**>get *filename* <*target-filename*>**
binary transport file	**>binary** **>get *filename* <*target-filename*>**

SASDOC-L

SASDOC-L is a listserv maintained by the Publications Division at SAS Institute. As a subscriber, you can request ASCII files that contain sample programs.

To use SASDOC-L, send e-mail, with no subject, to LISTSERV@VM.SAS.COM. The body of the message should be one of the lines listed below.

To subscribe to SASDOC-L, send this message:

SUBSCRIBE SASDOC-L
  <firstname lastname>

To get general information about files, download the file INFO by sending this message:

GET INFO EXAMPLES SASDOC-L

To get a list of available sample programs, download the file INDEX by sending this message:

GET INDEX EXAMPLES SASDOC-L

Once you know the name of the file you want, send this message:

GET *filename* EXAMPLES SASDOC-L

World Wide Web

The SAS Institute World Wide Web information server can be accessed at the following URL:

http://www.sas.com/

The sample programs are available from the Support Services portion of the Institute's server.

Syntax

This book uses the following syntax conventions:

`set search` *file-tag* = | + | – "*template1*" ["*template2*", . . .]

`unix2mf` [*option*, . . .]

`sascc370` [*options*] [*filename1* [*filename2*, . . .]]

`au` {*to*}

1 Commands, keywords, program names, and elements of the C language appear in monospace type.
2 Values that you must supply appear in italic type.
3 Mutually exclusive choices are joined with a vertical bar(|).
4 Optional arguments appear inside square brackets ([ ]).
5 Argument groups that you can repeat are indicated by an ellipsis (. . .).
6 Abbreviations are shown by curly braces ({ }).

Portability

This book uses the following icons to indicate the portability of functions:

ISO/ANSI Conforming

These functions conform to the ISO and ANSI C Language standards.

POSIX.1 Conforming

These functions conform the the POSIX.1 standard.

UNIX Compatible

These functions are commonly found in traditional UNIX C libraries.

SAS/C Extensions

These functions are not portable.

Additional Documentation

For a complete list of SAS publications, you should refer to the current *Publications Catalog*. The catalog is produced twice a year. You can order a free copy of the catalog by writing, calling, or faxing the Institute:

SAS Institute Inc.
Book Sales Department
SAS Campus Drive
Cary, NC 27513
Telephone: 919-677-8000 then press 1-7001
Fax: 919-677-4444
E-mail: sasbook@unx.sas.com

Online Documentation

This book is also available in html format for online viewing. See your SAS/C Software Consultant for information on accessing this book online.

SAS/C Software Documentation

In addition to *SAS/C Library Reference, Third Edition, Volume 1, Release 6.00*, you will find these other documents helpful when using SAS/C software:

- *SAS/C C++ Development System User's Guide, First Edition* (order #A56122) documents the SAS/C C++ Translator.
- *SAS/C CICS User's Guide, Second Edition, Release 6.00* (order #A55117) documents the SAS/C CICS Command Language Translator and the CICS version of the SAS/C Library.
- *SAS/C Compiler and Library User's Guide, Fourth Edition, Release 6.00* (order #A55156) provides a functional description of the SAS/C Compiler and is a reference for linking and executing C programs under TSO, CMS, and MVS.
- *SAS/C Compiler Interlanguage Communication Feature User's Guide* (order #A5684) documents the Interlanguage Communication Feature of the SAS/C Compiler.
- *SAS/C Cross-Platform Compiler and C++ Development System: Usage and Reference, First Edition, Release 6.00* (order #A55388) documents the cross-platform compiler and the C++ development system.
- *SAS/C Debugger User's Guide and Reference, Third Edition* (order #A56120) provides complete documentation for the SAS/C Debugger.
- *SAS/C Library Reference, Third Edition, Volume 2, Release 6.00* (order #A55178) describes the commonly used SAS/C Library functions.
- *SAS/C Full-Screen Support Library User's Guide, Second Edition* (order #A56124) documents the Full-Screen Support Library.
- *SAS/C Software Diagnostic Messages, First Edition, Release 6.00* (order #A55184) documents the SAS/C software diagnostic messages.
- *SAS/C Compiler and Library Quick Reference Guide, First Edition, Release 6.00* (order #A55182) provides quick reference information for the SAS/C Compiler, Library, and Debugger.
- *SAS/C Software: Changes and Enhancements to the SAS/C Debugger and C++ Development System, Release 6.00* (order #A55183) describes the Release 6.00 changes and enhancements that affect the SAS/C C++ Development System and the SAS/C Debugger.
- SAS Technical Report C-114, *A Guide for the SAS/C Compiler Consultant* (order #A59019) informs the SAS/C Software Consultant about the services provided by SAS Institute for SAS/C Compiler sites.
- SAS Technical Report C-115, *The Generalized Operating System Interface for the SAS/C Compiler Run-Time System, Release 5.50* (order #A59025) describes the Generalized Operating System Interface, which lets users write routines that enable the compiler's run-time library to access operating system services.

Supplementary Documentation

The following supplementary reference documentation is also recommended:

- Comer, Douglas E. (1991), *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture, Second Edition*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Comer, Douglas E., and Stevens, David L. (1993), *Internetworking with TCP/IP, Volume 3: Client-Server Programming and Applications, BSD Socket Version*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Stevens, W. Richard (1990), *UNIX Network Programming*, Englewood Cliffs, NJ: Prentice-Hall, Inc.
- Zlotnick, Fred (1991), *The POSIX.1 Standard: A Programmer's Guide*, Redwood City, CA: The Benjamin/Cummings Publishing Company, Inc.

Part 1

Using the SAS/C® Library

Chapters

1 Introduction to the SAS/C® Library

Commonly Used Functions

This book describes the most commonly used functions in the SAS/C library. Chapter 2, "Function Categories," itemizes the functions by the following category:

- character type macros and functions
- string utility functions
- mathematical functions
- varying-length argument list functions
- general utility functions
- program control functions
- memory allocation functions
- diagnostic control functions
- timing functions
- I/O functions
- file management functions
- system interface and environment variables functions
- signal-handling functions.

Functions are listed alphabetically in Chapter 6, "Function Descriptions."

Special Features

The following special features of the SAS/C library are documented in *SAS/C Library Reference, Volume 2*:

- dynamic-loading functions
- CMS low-level I/O functions
- MVS low-level I/O functions
- low-level system interfaces
- MVS low-level multitasking functions
- inter-user communications vehicle (IUCV) functions
- advanced program-to-program communication/virtual machine (APPC/VM) functions
- the subcommand interface to EXECs and CLISTs

- □ the CMS REXX SAS/C interface
- □ coprocessing functions
- □ localization functions
- □ multibyte character functions.

Additionally, *SAS/C Library Reference, Volume 2* documents the SAS/C socket library for TCP/IP and SAS/C POSIX support.

Compatibility with Standards

Most functions in the SAS/C library are compatible with industry-recognized C library standards, including

- □ ISO/ANSI C
- □ traditional UNIX C libraries
- □ POSIX 1003.1.

ISO/ANSI C Standards

The SAS/C library is fully compliant with the 1990 ISO/ANSI C standard. Implementation-defined behavior for the ISO/ANSI library is described in Chapter 2, "Language Definition" in *SAS/C Compiler and Library User's Guide.*

Traditional UNIX Support

The SAS/C library supports a number of functions defined by traditional (pre-POSIX) UNIX systems. In some cases, these functions are limited to use with the OpenEdition MVS operating system, an International Business Machines Corporation Product. However, in many cases, these functions have been defined so that they are meaningful in native MVS and CMS environments. For instance, although the **`stat`** and **`link`** functions are limited to use with OpenEdition files, other functions such as **`open`**, **`read`**, **`write`**, and **`access`** can be used with most MVS and CMS file types.

Unlike the ISO/ANSI and POSIX libraries, the traditional UNIX library is not defined as a formal standard. Rather, the traditional UNIX library is informally defined by consensus with a number of different UNIX implementations, based on both System V and BSD.

Although SAS/C does not and cannot support every function defined by every historical UNIX variant, the library does attempt to offer support for a large subset of the core UNIX functionality, especially functions frequently used in portable programs. Note, however, that some core functions, such as **`fork`** and **`kill`**, cannot be implemented under MVS or CMS by an application-level library such as the SAS/C library without operating system support (such as OpenEdition MVS).

POSIX Standards

In addition to the functionality provided in earlier releases of SAS/C, Release 6.00 supports OpenEdition MVS. OpenEdition comprises three products:

- □ MVS support for OpenEdition
- □ the Shell and Utilities product
- □ the dbx debugger.

SAS/C, Release 6.00 directly uses the MVS support for OpenEdition, and you can use it with the Shell and Utilities product. The dbx debugger does not support SAS/C programs, but you can use the traditional SAS/C debugger under the shell instead of using dbx.

Using the underlying functionality of the MVS support for OpenEdition, SAS/C, Release 6.00 enables you to

- □ write programs using OpenEdition functionality using interfaces defined by the POSIX 1003.1 and 1003.1a standards

- run programs compiled with SAS/C under the OpenEdition shell
- use the SAS/C debugger to debug programs under the OpenEdition shell.

To support these features, the Institute made some changes to the SAS/C compiler and debugger, but most changes are localized to the resident and transient libraries. Compile-time header files are also significantly changed.

The POSIX 1003.1 standard is an ISO standard that specifies operating system functionality in a C language interface. With OpenEdition, the SAS/C library implements this interface under MVS. OpenEdition and SAS/C also implement portions of the 1003.1a draft standard and related extensions. POSIX 1003.1 is based on common elements of a number of UNIX operating systems.

The SAS/C POSIX implementation is documented in Part 3, "SAS/C POSIX Support," in *SAS/C Library Reference, Volume 2.*

Rules for Using Different Releases of the Compiler and Library

Here are the rules for compiling, linking, and executing programs with different releases of the compiler and library:

- For a newly compiled program, use a more recent release of the transient library than the compiler, or use the same release. A program compiled and linked with an older release will run with a new release of the transient library. If you run a newly compiled and linked program with an old release of the transient library, the program may fail in various ways, possibly with an 0C1 or 0C6 ABEND.
- Use a release of the transient library that is the same or more recent than the resident library. If the transient library detects a release mismatch with the resident library, it prints a warning message.
- Use a release of the resident library that is the same or more recent than the compiler. If you link compiled code with a wrong release of the resident library, no warning is produced. (However, you will likely get a system 0CX ABEND when you execute the program.)
- Use a version of the SAS/C header files that is no more recent than the compiler version.
- To link code produced by various releases of the compiler, use a release of the resident library that is at least as recent as the most recent release of the compiler used.
- For an application with multiple load modules, link all modules of the load module with the same release of the resident library. If you do not, unpredictable errors may occur. However, you can still use more than one level of the compiler to generate the object code, provided that the modules are all compatible with the level of the resident library as specified earlier in this section.

Table 1.1 shows the likely result of using different releases of the compiler and the resident and transient libraries to compile and link programs. If you are unsure which version of the library you are using, you can use the `=version` run-time option, which displays the library version numbers.

Note: Combining more than three versions of the compiler and resident and transient libraries becomes very complicated and is not documented here. In Table 1.1 where a row contains two entries of "older," they refer to the *same* older version.

Table 1.1
Likely Results of Mixing Releases of the Compiler and Libraries

Program	Transient Library Release	Resident Library Release	Compiler Release	Likely Result: Will Program Run?
1	current	current	current	Yes
2	current	current	previous	Yes
3	current	current	older	Yes
4	current	previous	current	No
5	current	previous	previous	Yes
6	current	previous	older	Yes
7	current	older	current	No
8	current	older	previous	No
9	current	older	older	Yes
10	previous	current	current	No
11	previous	current	previous	No
12	previous	current	older	No
13	previous	previous	current	No
14	previous	previous	previous	Yes
15	previous	previous	older	Yes
16	previous	older	current	No
17	previous	older	previous	No
18	previous	older	older	Yes
19	older	current	current	No
20	older	current	previous	No
21	older	current	older	No
22	older	previous	current	No
23	older	previous	previous	No
24	older	previous	older	No
25	older	older	current	No
26	older	older	previous	No
27	older	older	older	Yes

Library Header Files

The functions provided by the library are associated with header files. Each header file contains the function prototype and any necessary types and macros associated with the functions. In some cases, the correct use of a function may require more than one header file.

For maximum portability and efficiency, always include the header files for all of the library functions called in a compilation. This practice has two benefits:

- The function prototype is in scope when the program is compiled, enabling the compiler to flag incorrect or potentially nonportable uses of the function.
- If the function is implemented as a macro or as a built-in function, the header file will have the correct macro definition. It is always more efficient to use the macro or built-in version of a function than to use the true function. (For more information on built-in functions, refer to "Built-in Functions and Macros" on page 1-15.)

Header File Organization

The SAS/C library defines a strict separation of functions into three parts:

- functions defined by the ISO/ANSI standard
- functions defined by another standard, such as the POSIX.1 standard
- common nonstandard functions or SAS/C extensions.

Functions defined by the ISO/ANSI standard are declared in the header file mandated by the standard. For example, the **fopen** function is declared in **<stdio.h>**. The names of the standard header files are

```
<assert.h>  <setjmp.h>
 <ctype.h>  <signal.h>
 <errno.h>  <stdarg.h>
 <float.h>  <stddef.h>
<limits.h>  <stdio.h>
 <local.h>  <stdlib.h>
  <math.h>  <string.h>
  <time.h>
```

Functions defined by another standard are declared in the header file mandated by the standard. If that file is also an ISO/ANSI C standard header file, you must use a feature test macro to make the declaration visible. Feature test macros are described in more detail in the next section.

Functions which are not defined by the ISO/ANSI Standard but which are related to standard functions are declared in separate header files. These header filenames are similar to the Standard names but have the prefix **lc**. For example, the function **afopen** is declared in **<lcio.h>**. The names of these header files are

```
<lcdef.h>  <lcmath.h>
 <lcio.h>  <lcsignal.h>
<lcjmp.h>  <lcstring.h>
<lclib.h>  <lctype.h>
```

This separation of functions is intended as an aid in writing portable programs. Only those functions declared in the standard header files are completely portable.

If you include the **lc-** header file, you do not need to include the standard header file. In all cases, the **lc-** prefixed header contains a **#include** statement for the standard header file. For example, the header file **<lcstring.h>** contains the statement **#include <string.h>**. (It is not an error to explicitly include both files.)

The SAS/C library contains many nonstandard functions and header files that are not associated with standard features. For details on nonstandard functions and header files that are not associated with standard features, see *SAS/C Library Reference, Volume 2.*

Feature Test Macros

Feature test macros are defined by various IEEE POSIX standards to enable you to specify the standards and language features that you wish to use. SAS/C uses feature test macros in the following way:

- □ An ISO/ANSI standard header file contains only declarations permitted by the ISO/ANSI C standard, unless the user defines an appropriate feature test macro before including the file.
- □ A non-ISO/ANSI header file normally contains declarations for both standard and nonstandard features. However, by using appropriate feature test macros you can cause unwanted extensions to be unavailable.

SAS/C supports the following feature test macros. To enable a feature, you must define the macro before including any header file, either by using a **`#define`** statement or by using the **`define`** compiler option.

`_SASC_POSIX_SOURCE`
: If this macro is defined as any value, symbols defined by a supported POSIX.1 standard will be made visible in ISO/ANSI standard header files. **`_SASC_POSIX_SOURCE`** has no effect on non-ISO/ANSI header files.

`_POSIX_SOURCE`
: If this macro is defined as any value, symbols defined by the POSIX.1 standard will be made visible in ISO/ANSI standard header files. Also, declarations of any symbols that are not specified as allowable in POSIX.1-header files in the POSIX.1 standard will be suppressed. **`_POSIX_SOURCE`** should be defined only for programs which are intended to be POSIX-conforming and which do not use any non-ISO/ANSI or non-POSIX library features.

`_POSIX1_SOURCE`
: If this macro is defined as 1, the effect is the same as defining **`_POSIX_SOURCE`**. If this macro is defined as 2, symbols sanctioned by the POSIX.1a draft standard related to features implemented by OpenEdition MVS will also be made visible. Like **`_POSIX_SOURCE`**, **`_POSIX1_SOURCE`** should not be defined in any program that uses non-ISO/ANSI non-POSIX features.

`_POSIX_C_SOURCE`
: If this macro is defined as 2, it has the effect of defining **`_POSIX_SOURCE`**, plus making visible symbols sanctioned by the POSIX.2 draft standard related to features implemented by SAS/C. If **`_POSIX_C_SOURCE`** is defined to any other value, it has the same effect as defining **`_POSIX_SOURCE`**. Like **`_POSIX_SOURCE`**, **`_POSIX_C_SOURCE`** should not be defined in any program that uses non-ISO/ANSI non-POSIX features.

If you use the POSIX compiler option, the feature test macro **`_SASC_POSIX_SOURCE`** is automatically defined. This does not ensure that your program is POSIX compliant; it only makes visible POSIX symbols in ISO/ANSI standard header files.

The errno Variable

The external `int` variable `errno` contains the number of the most recent error or warning condition detected by the run-time library. To use this value, include the header file `<errno.h>`.

If no error or warning condition is detected, the value of `errno` is 0. After program execution starts, `errno` is never reset to 0 by the library. Programs that use `errno` for information about unusual conditions must set it to 0 before calling a library routine that may detect such a condition.

The `<errno.h>` file contains declarations of the `errno` variable and definitions of symbolic names for the values that can be assigned. These names rather than numeric values should be used for `errno`.

SAS/C defines a number of general-use `errno` names. There are also many `errno` names associated with specific sublibraries, notably the SAS/C socket library and the SAS/C POSIX support. Socket `errno` names are documented in Chapter 15, "The BSD UNIX Socket Library," in *SAS/C Library Reference, Volume 2*, and `errno` names related to OpenEdition are documented in Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2.* For a complete listing of all `errno` values, see *SAS/C Compiler and Library Quick Reference Guide.*

The following list defines the error names and meanings that are for general use, and thus not associated with a specialized API, such as sockets:

`EARG`	undefined function argument value
`EBADF`	file or socket not open or suitable (synonym for `ENOTOPEN`)
`ECONV`	data conversion failure
`ECORRUPT`	file is in a corrupt or unreadable state
`EDEVICE`	physical device error
`EDOM`	math function domain error
`EDUPKEY`	attempt to add record with duplicate key
`EEXIST`	file already exists
`EFATTR`	file attribute conflict
`EFFORM`	file format error
`EFORBID`	function execution prevented by run-time options
`EILSEQ`	error in multi-byte character sequence (reserved for future use)
`EINTR`	function failed due to interruption by signal
`EINUSE`	file to be opened was already in use
`EINVAL`	invalid argument (synonym for `EARG`)
`EIO`	physical I/O error (synonym for `EDEVICE`)
`ELIBERR`	run-time system internal error
`ELIMIT`	internal limit exceeded
`EMFILE`	too many open files (synonym for `ELIMIT`)
`ENFILE`	too many open HFS files in system
`ENFOUND`	file not found
`ENOENT`	file or directory not found (synonym for `ENFOUND`)
`ENOMEM`	insufficient memory
`ENOSPC`	no space in file
`ENOSYS`	function not implemented by system

ENOTOPEN	synonym for **EBADF**
EPREV	previous error not cleared
ERANGE	math function range error
ESYS	operating system interface failure
EUNSUPP	unsupported I/O operation
EUSAGE	incorrect function usage.

The variable **errno** is implemented as a macro. If you use **errno** without including **<errno.h>**, the correct data may not be accessed.

The only portable values for **errno** are **EDOM** and **ERANGE**. The following example illustrates the use of **errno**:

```
#include <errno.h>
#include <stdio.h>

FILE *f;
char *filename;
if (!(f = fopen(filename, "r"))) {
      /* See if any file can be opened. */
   if (errno == ELIMIT) {
      printf("Too many open files\n");
      return(EOF);
   }
   else {
      printf("%s could not be opened, enter new filename\n",
             filename);
      getfname(filename);
   }
}
```

More Exact Error Information: _msgno Variable

The header file **<lcdef.h>** contains the declaration of a nonstandard external variable, **_msgno**. This variable contains the message number of the last SAS/C library diagnostic. For example, if the last message ID were LSCX503, **_msgno** would contain 503.

_msgno may provide more information about a failure than **errno**. For instance, trying to read a file that has not been created sets **errno** to ENFOUND, but you can use **_msgno** to distinguish the cases of an empty sequential file (**_msgno** = 503) and a missing PDS member (**_msgno** = 504). **_msgno** is not portable, so programs that must be portable should use only **errno**.

_msgno is implemented as a macro, so you should not use the name for any other purpose.

System Macro Information

The SAS/C System Macro Information (SYSMI) facility provides a way for a program to determine accurate information about a library failure caused by an error return code from a system macro or service. The information available includes the name of the service and the numeric codes associated with the failure.

When the library calls an operating system service (including an OpenEdition system call) that fails, information about the failure is saved in a library structure. Macros are defined in **<lcdef.h>** to enable user code to determine the service that failed and the resulting failure codes. Only the most recent failure information is saved; information is not saved for successful services.

The following macros are defined:

`__sysmi_macname`
expands to a null-terminated string naming the macro or service that failed. For OpenEdition system calls, this is the BPX name of the failing service.

`__sysmi_rc`
is the return code of the failing service. For OpenEdition system calls, this is the numeric value returned by OpenEdition before library translation into an **`errno`** value.

`__sysmi_reason`
is the reason code of the failing service (or 0 if no reason code was returned). For OpenEdition system calls, you can find the meaning of the last two bytes of the reason code in the IBM publication *Assembler Callable Services for OpenEdition MVS*.

`__sysmi_info`
is the information code for the failing service (or 0 if this code is not applicable).

You can use the macro **`__sysmi_clear`** to clear previously stored system macro information. You may wish to call **`__sysmi_clear`** before calling a routine that might store SYSMI information to ensure that any such information relates to the most recently called function.

Definitions: <lcdef.h>

Several nonstandard macros are defined in **`<lcdef.h>`**. The following pages describe these macros, **`offsetof`**, **`isunresolved`**, **`isnotconst`**, **`isnumconst`**, and **`isstrconst`**.

offsetof Get the Byte Offset of a Structure Component

SYNOPSIS

```
#include <stddef.h>

   /* macro */
size_t offsetof(type, element)
```

DESCRIPTION

The **`offsetof`** macro provides the decimal byte offset of a component within a structure as a **`size_t`** constant. This constant is generated at compile time. Padding for alignment, if any, is included. The operands of **`offsetof`** are a structure type (**`type`**) and a component of the structure specification (**`element`**). The component specification does not include the structure type or the selection operators **`.`** or **`->`**.

RETURN VALUE

`offsetof` returns the byte offset of **`element`**.

EXAMPLES

As shown in these examples, you should write the member specification as it would be written to access the value of a structure member, except that there is no leading **`.`** or **`->`** selection operator.

Example 1.1

```
#include <stddef.h>

struct AAA {    /* Define structure AAA. */
   double ddd;
   char ccc;
   int bbb;
   };
long x;
   /* x is the byte offset of component bbb in struct AAA. */
x = offsetof(struct AAA, bbb);
```

Example 1.2 shows a structure, **`data`**, with an inner structure **`base`**.

Example 1.2

```
#include <stddef.h>

struct data {        /* Define struct data.      */
   int id;
   int *elem;
   char *name;
   struct {          /* Define struct type base. */
      double proj;
      } base;
   };
```

offsetof Get the Byte Offset of a Structure Component
(continued)

```
long ofs;
   /* ofs is the byte offset of base.proj.     */
ofs = offsetof(struct data, base.proj);
```

In Example 1.3, **`complex`** is defined via a **`typedef`** statement to be a structure type. The component specification **`inner.d[5]`** specifies an array element within an inner structure. The variable **`y`** is set to the offset of the sixth array element in the inner structure (**`decimal 56`**).

Example 1.3

```
#include <stddef.h>

typedef struct {   /* Define struct type complex. */
   struct XXX *xptr, *xptr2;
   struct {         /* Define struct type inner.  */
      int count, count2;
      double d[10];
      } inner;
   struct XXX *xptr3;
   } complex;
      /* y is the byte offset of inner.d[5].       */
   long y;
   y = offsetof(complex, inner.d[5]);
```

isunresolved Test Whether an External Symbol is Resolved

SYNOPSIS

```
#include <lcdef.h>

int isunresolved(name);
```

DESCRIPTION

This macro tests the name (which should be the name of a variable declared as **`__weak`**) to determine whether the symbol was resolved by the linkage editor.

RETURN VALUE

`isunresolved` returns 0 if the symbol is resolved, or a nonzero value if it is not resolved.

EXAMPLE

```
/* Test whether the function db_open() is present */
/* in the load  module.  If it is, call it.       */

   /* optional database open function */
extern int __weak db_open(char *);

if (!isunresolved(db_open))
   db_open("DBNAME");
```

SEE ALSO

Chapter 2, “Language Definition,” in *SAS/C Compiler and Library User’s Guide*

isnotconst	Test for Nonconstant
isnumconst	Test for Numeric Constant
isstrconst	Test for String Literal

SYNOPSIS

```
#include <lcdef.h>
int isnotconst(expression);
int isnumconst(expression);
int isstrconst(expression);
```

DESCRIPTION

These macros examine **`expression`** and return a compile-time constant. If **`expression`** is the appropriate type of constant, a nonzero constant is returned; otherwise, 0 is returned. The type tested for is numeric for **`isnumconst`**, string literal for **`isstrconst`**, and nonconstant for **`isnotconst`**. The **`expression`** constant can have any type.

`expression` is never evaluated, and these macros always yield a constant, regardless of the type of **`expression`**.

The **`isnotcons`**, **`isnumconst`**, and **`isstrconst`** macros are used primarily to control the generation of code by in-line functions. Because they produce compile-time constants, the macros can be tested at compile time, enabling the compiler to eliminate sections of code that can never be executed.

EXAMPLES

Below are several examples using these nonstandard macros:

Example 1.4

```
if (isnotconst(argv[]))               /* true  */
if (isnumconst(100))                  /* true  */
if (isstrconst("XYZ"))                /* true  */
if (isstrconst(c == 0 ? "A" : "B")    /* false */
```

Example 1.5

```
#define MAXLEN 1024

if (isnumconst(MAXLEN) && 500 < MAXLEN) /* true  */
```

Example 1.6

This example defines the function **`smemcpy`** (meaning short **`memcpy`**) that prevents the expansion of the built-in **`memcpy`** function unless the length argument is a constant integer less than or equal to 256. If the length argument is greater than 256 or is not a constant integer, a call to the true **`memcpy`** function is generated.

The **`if`** condition is a constant expression and is evaluated at compile time. The compiler generates code either for the **`then`** branch or the **`else`** branch, depending on the result of the test. Under no conditions is code for both branches generated.

isnotconst
isnumconst
isstrconst
(continued)

Test for Nonconstant

```
#include <lcdef.h>
#include <string.h>

#define smemcpy(d, s, len)
        inline_memcpy(d, s, len, isnumconst(len))
__inline
void *inline_memcpy(void *d, const void *s,
                    size_t len, int cnst)
{
   if (cnst && len < 257)
      memcpy(d, s, len);
   else
      (memcpy)(d, s, len);
   return d;
}
```

Implementation of Functions

Built-in Functions and Macros

Many of the functions in the library are implemented as built-in functions. A *built-in function* is a function for which the compiler generates the required machine instructions directly in the compiled code instead of making a call to a separately compiled routine. *True functions* are compiled separately and must be linked with the program before they can be executed. By eliminating the overhead of parameter list creation and branching, a built-in function is always more efficient than a call to a true function. Generally, built-in functions can be implemented by a relatively short sequence of machine instructions. These afford the greatest increase in efficiency. The **abs** function is a good example:

```
#include <math.h>
int gt5(register int i){
 return (i < -5) ? i + 5 : abs(i);
 }
```

Given this C function, the compiler generates a single IBM 370 machine instruction called Load Positive Register (LPR) to get the absolute value of **i**. However, calling and executing the true **abs** function in this example requires the execution of 20 machine instructions.

The compiler and library implement built-in functions by defining a macro in the header file that prefixes the string **__builtin_** to the function name. For example, the **strcpy** function is declared as follows:

```
#define strcpy(x, y) __builtin_strcpy(x, y)
```

The compiler recognizes the prefix and generates the appropriate machine instructions. If you do not include the header file, the compiler does not recognize the function as a built-in function and generates a call for the function.

For some built-in functions, the compiler may generate a call to the true function as part of the code sequence. This occurs when the value of one or more of the function arguments cannot be determined at compile time and may fall outside of the range of values that the in-line code can handle. At execution time, the arguments are evaluated, and either the in-line code is executed or the true function is called.

If a built-in function is called with invalid arguments or an invalid number of arguments, a call to the true function is generated.

Following is a list of all SAS/C built-in functions:

_bbwd	_stregs	memcmpp	strcmp
_bfwd	abs	memcpy	strcpy
_branch	ceil	memcpyp	strlen
_cc	fabs	memscan	strncmp
_cms202	floor	memscntb	strscan
_code	fmax	memset	strscntb
_diag	fmin	memxlt	strxlt
_label	getc	min	tolower
_ldexp	labs	modf	toupper
_ldregs	max	putc	
_ossvc	memcmp	sigchk	

Following is a list of all SAS/C functions implemented as macros, other than built-in functions:

ABEND*	TPUT_ASID*	fnm+	onjmp*
CHAP*	TPUT_USERID*	fterm+	onjmpout*
CMSSTOR_OBT*+	TTIMER*	getchar	palloc
CMSSTOR_REL*+	WAIT1*	htonl	pause
DEQ*	WAITM*	htons	pdset*
DETACH*	WAITT*	isalnum	pdval*
DMSFREE*	WRTERM*	isalpha	pfree
DMSFREE_V*	appcconn*+	iscics	putchar
DMSFRET*	appcrecv*+	iscntrl	setbuf
ENQ*	appcscfd*+	iscsym	shvdrop*
ESTAE*	appcscnf*+	iscsymf	shvfetch*
ESTAE_CANCEL*	appcsdta*+	isdigit	shvfirst*
FREEMAIN*	appcserr*+	isebcdic	shvnext*
GETLONG*	appcsevr*+	isgraph	shvset*
GETMAIN_C*	appcsreq*+	islower	sigsetjmp*
GETMAIN_V*	assert*	isnotconst*	strcspn
GETMAIN_U*	atof	isnumconst*	strspn
GETSHORT*	atoi	isprint	toebcdic
POST*	cfgetispeed	ispunct	typlin*
PUTLONG*	cfgetospeed	isspace	unloadd
PUTSHORT*	cmspush*	isstrconst*	va_arg*
RDTERM*	cmsqueue*	isunresolved*	va_end
STATUS*	difftime	isupper	va_start*
STIMER*	e_SVC202	isxdigit	waitrd*+
STIMERM_CANCEL*+	execdrop*	labs	xedpoint
STIMERM_SET*	execfetch*	localtime	xedread
STIMERM_TEST*	execset*	memchr	xedstate
SVC202	exit	ntohl	xedwrite
TGET*	fattr+	ntohs	
TPUT*	ffixed+	offsetof*	

* = may not be undefined.
+ = evaluates some arguments more than once.

2 Function Categories

Introduction

This chapter briefly describes these function categories:

- □ character type macros
- □ string utility
- □ mathematical
- □ varying-length argument list
- □ general utility
- □ program control
- □ memory allocation
- □ diagnostic control
- □ timing
- □ I/O
- □ file management
- □ system interface and environment variables
- □ signal-handling.

Character Type Macros and Functions

The character type header files, **`<ctype.h>`** and **`<lctype.h>`**, define several macros that are useful in the analysis of text data. Most of these macros enable you to determine quickly the type of a character (whether it is alphabetic, numeric, punctuation, and so on). These macros refer to an external array that is indexed by the character itself, so they are generally much faster than functions that check the character against a range or discrete list of values. Note that this array is actually indexed by the character value plus 1, so the standard **`EOF`** value (-1) can be tested in a macro without yielding a nonsense result. **`EOF`** yields a 0 result for all of the macros because it is not defined as any of the character types. Also, note that the results produced by most of these functions are affected by the current locale's **`LC_CTYPE`** category, which may cause a different character type array to be used than the one supplied for the default C locale. See Chapter 10, “Localization,” in *SAS/C Library*

Reference, Third Edition, Volume 2, Release 6.00 for details on locales.

Another advantage of the character type macros is that they prevent problems when programs are moved between machines that use ASCII versus EBCDIC character sets. Programs using these macros are not dependent on a specific character set.

The following are character type macros and functions:

`isalnum`	alphanumeric character test
`isalpha`	alphabetic character test
`isascii`	ASCII character test
`iscntrl`	control character test
`iscsym`	test for valid C identifier symbol
`iscymf`	test for valid C identifier initial symbol
`isdigit`	test for numeric character
`isebcdic`	EBCDIC character test
`isgraph`	graphic character test
`islower`	lowercase alphabetic character test
`isprint`	printing character test
`ispunct`	punctuation test
`isspace`	white space test
`isupper`	uppercase alphabetic character test
`isxdigit`	hexadecimal digit test
`toebcdic`	reduce integer to EBCDIC character
`tolower`	translate uppercase character to lowercase
`toupper`	translate lowercase character to uppercase.

Table 2.1 lists the macros defined in the character type header files **`<ctype.h>`** and **`<lctype.h>`**. The library conforms to the ISO/ANSI specification in that the macro arguments are evaluated only once. However, many implementations do not conform to this specification. For maximum portability, beware of the side effects of using expressions such as function calls and increment or decrement operators. You should include **`<ctype.h>`** or **`<lctype.h>`** if you use any of these macros; otherwise, the compiler generates a reference to a function of the same name.

Table 2.1
Character Type Macros and Functions and Their Return Values

Function	Return Value
`isalnum(c)`	nonzero if **`c`** is alphabetic or digit; 0 if not
`isalpha(c)`	nonzero if **`c`** is alphabetic; 0 if not
`isascii(c)`*	nonzero if **`c`** is the EBCDIC equivalent of an ASCII character; 0 if not
`iscntrl(c)`	nonzero if **`c`** is control character; 0 if not
`iscsym(c)`*	nonzero if valid character for C identifier; 0 if not

continued

Table 2.1 (*continued*)

Function	**Return Value**
`iscsymf(c)`	nonzero if valid first character for C identifier; 0 if not
`isdigit(c)`*	nonzero if `c` is a digit 0–9; 0 if not
`isebcdic(c)`*	nonzero if a valid EBCDIC character; 0 if not
`isgraph(c)`	nonzero if `c` is graphic (excluding the blank character); 0 if not
`islower(c)`	nonzero if `c` is lowercase; 0 if not
`isprint(c)`	nonzero if `c` is printable (including blank); 0 if not
`ispunct(c)`	nonzero if `c` is punctuation; 0 if not
`isspace(c)`	nonzero if `c` is white space; 0 if not
`isupper(c)`	nonzero if `c` is uppercase; 0 if not
`isxdigit(c)`*	nonzero if `c` is a hexadecimal digit (0–9, A–F, a–f); 0 if not
`toebcdic(c)`*	truncates integer to valid EBCDIC character
`tolower(c)`	converts `c` to lowercase, if uppercase
`toupper(c)`	converts `c` to uppercase, if lowercase

*These functions are not affected by the locale's `LC_TYPE` category.

Note: The `toupper` and `tolower` macros generate the value of `c` unchanged if it does not qualify for the conversion.

String Utility Functions

The C library provides several functions to perform many string manipulations. There are three general categories of string utility functions:

functions that begin with the letter `a`
: convert character strings to numbers.

functions that begin with the letters `str`
: treat their arguments as strings that are terminated with a null character.

functions that begin with the letters `mem`
: treat their arguments as byte strings in which a null character is not considered a terminator. The `mem` routines are always passed an explicit string length since the string may contain no null characters, or more than one null character.

Two standard string functions that begin with the letters `str`, `strcoll`, and `strxfrm` pertain to localization and are discussed in Chapter 10, "Localization," in *SAS/C Library Reference, Volume 2.*

The following are string functions:

`atof`	convert a string to floating point
`atoi`	convert a string to integer

atol	convert a string to long
memchr	locate first occurrence of a character
memcmp	compare two blocks of memory
memcmpp	compare two blocks of memory with padding
memcpy	copy characters
memcpyp	copy characters (with padding)
memfil	fill a block of memory with a multicharacter string
memlwr	translate a memory block to lowercase
memmove	copy characters
memscan	scan a block of memory using a translate table
memscntb	build a translate table for use by **memscan**
memset	fill a block of memory with a single character
memupr	translate a memory block to uppercase
memxlt	translate a block of memory
stcpm	unanchored pattern match
stcpma	anchored pattern match
strcat	concatenate two null-terminated strings
strchr	locate first occurrence of a character in a string
strcmp	compare two null-terminated strings
strcpy	copy a null-terminated string
strcspn	locate the first occurrence of the first character in a set
strlen	compute length of null-terminated string
strlwr	convert a string from uppercase to lowercase
strncat	concatenate two null-terminated strings (limited)
strncmp	compare portions of two strings
strncpy	copy a limited portion of a null-terminated string
strpbrk	find first occurrence of character of set in string
strrchr	locate the last occurrence of a character in a string
strrcspn	locate the last character in a set
strrspn	locate the last character of a search set not in a given set
strsave	allocate a copy of a character string
strscan	scan a string using a translate table
strscntb	build a translate table for use by **strscan**
strspn	locate the first occurrence of the first character not in a set
strstr	locate first occurrence of a string within a string
strtod	convert a string to double
strtok	get a token from a string
strtol	convert a string to long integer
strtoul	convert a string to an unsigned long integer
strupr	convert a string from lowercase to uppercase

`strxlt`	translate a character string
`xltable`	build character translation table.

Terms Used in String Function Descriptions

These terms are used in the descriptions of string utility functions:

string	is zero or more contiguous characters terminated by a null byte. The first character of a string is at position 0. Functions that return the `int` or `unsigned` position of a character in a string compute the position beginning at 0.
character sequence	is a set of contiguous characters, not necessarily null-terminated.

Optimizing Your Use of memcmp, memcpy, and memset

You can optimize your use of the built-in functions `memcmp`, `memcpy`, and `memset` by controlling the type of the `length` argument. The compiler inspects the type before the argument is converted to the type specified in the function prototype. If the type of the `length` argument is one of the types in Table 2.2, the compiler generates only the code required for the maximum value of the type. Table 2.2 shows the maximum values of these types. Note that these values can be obtained from the `<limits.h>` header file.

You can use only the types shown in Table 2.2 (in addition to `size_t`). If the `length` argument has any other type, the compiler issues a warning message.

Table 2.2
Types Acceptable as Length Arguments in Built-in Functions

Type	Maximum Value
`char`	255
`unsigned char`	255
`short`	32767
`signed short`	32767
`unsigned short`	65535

If Table 2.2 lists the type of the `length` argument, the function will not be required to operate on more than 16 megabytes of data. Therefore, the compiler does not generate a call to the true (that is, separately linked) function to handle that case.

If the `length` argument is one of the `char` types, the compiler generates a MOVE instruction (which can handle up to 256 characters) rather than a MOVE LONG (which can handle up to 16 megabytes of characters). Because the MOVE LONG instruction is one of the slowest instructions in the IBM 370 instruction set, generating a MOVE saves execution time.

Getting the Most Efficient Code

To get the compiler to generate the most efficient code sequence for string functions, follow these guidelines:

1. Use the built-in version of the function. Built-in functions are defined as macros in the appropriate header file. Always include `<string.h>` or `<lcstring.h>`, and do not use the function name in an `#undef` preprocessing directive.
2. Declare or cast the `length` argument as one of the types in Table 2.2.
3. Do not cast the `length` argument to a wider type. This defeats the compiler's inspection of the type.

You may want to define one or more macros that cast the `length` argument to a shorter type. For example, here is a program that defines two such macros:

```
#include <string.h>

   /* Copy up to 32767 characters. */
#define memcpys(to, from, length) memcpy(to, from, (short)length)

   /* Copy up to 255 characters. */
#define memcpyc(to, from, length) memcpy(to, from, (char)length)
.
.
.
int strsz;          /* strsz is known to be less than 32K. */
char *dest, *src;

memcpys(dest, src, strsz);         /* casts strsz to short */
.
.
.
```

Some recent IBM processors include a hardware feature called the Logical String Assist, which implements the C functions **strlen**, **strcpy**, **strcmp**, **memchr**, and **strchr** in hardware. To make use of this hardware feature, **#define** the symbol **_USELSA** before including **<string.h>** or **<lcstring.h>**. The resulting code will not execute on hardware that does not have the Logical String Assist feature installed.

Mathematical Functions

The mathematical functions include a large proportion of the floating-point math functions usually provided with traditional UNIX C compilers. The header file **<math.h>** should be included when using most of these functions. See the individual function descriptions to determine whether the header file is required for that function.

The library also provides the standard header file **<float.h>**, which provides additional information about floating-point arithmetic. The contents of this header file are listed here:

```
#define FLT_RADIX 16          /* hardware float radix              */
#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 16            /* double decimal precision          */
#define LDBL_DIG 16           /* 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          */
#define DBL_MAX .72370055773322621e76     /* maximum double         */
#define LDBL_MAX .72370055773322621e76L   /* maximum long double    */

   /* smallest float x such that 1.0 + x != 1.0                     */
#define FLT_EPSILON .9536743e-6F

   /* smallest double x such that 1.0 + x != 1.0                    */
#define DBL_EPSILON .22204460492503131e-15

   /* smallest long double x such that 1.0 - x != 1.0               */
#define LDBL_EPSILON .22204460492503131e-15L

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

Additionally, the header file **`<lcmath.h>`** declares useful mathematical constants, as listed in Table 2.3.

Table 2.3
Constant Values Declared in lcmath.h

Constant	Representation
M_PI	π
M_PI_2	$\pi/2$
M_PI_4	$\pi/4$
M_1_PI	$1/\pi$
M_2_PI	$2/\pi$
M_E	e
HUGE*	largest possible double
TINY	double closest to zero
LOGHUGE	log(HUGE)
LOGTINY	log(TINY)

***`math.h`** defines the value **`HUGE_VAL`**, which is an ANSI-defined symbol with the same value.

The following are mathematical functions:

`abs`	integer conversion: absolute value
`acos`	compute the trigonometric arc cosine

asin	compute the trigonometric arc sine
atan	compute the trigonometric arc tangent
atan2	compute the trigonometric arc tangent of a quotient
ceil	round up a floating-point number
cos	compute the trigonometric cosine
cosh	compute the hyperbolic cosine
div	integer conversion: division
erf	compute the error function
erfc	compute the complementary error function
exp	compute the exponential function
fabs	floating-point conversion: absolute value
floor	round down a floating-point number
fmax	find the maximum of two doubles
fmin	find the minimum of two doubles
fmod	floating-point conversion: modules
frexp	floating-point conversion: fraction-exponent split
gamma	compute the logarithm of the gamma function
hypot	compute the hypotenuse function
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*
labs	integer conversion: absolute value
ldexp	floating-point conversion: load exponent
_ldexp	fast implementation of **ldexp**
ldiv	integer conversion: division
log	compute the natural logarithm
log10	compute the common logarithm
_matherr	handle math function error
max	find the maximum of two integers
min	find the minimum of two integers
modf	floating-point conversion: fraction-integer split
pow	compute the value of the power function
rand	simple random number generation
sin	compute the trigonometric sine
sinh	compute the hyperbolic sine
sqrt	compute the square root
srand	simple random number generation
tan	compute the trigonometric tangent
tanh	compute the hyperbolic tangent
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*.

Varying-Length Argument List Functions

This category of functions contains three macros that advance through a list of arguments whose number and type are unknown when the function is compiled. The macros are

`va_arg`	access an argument from a varying-length argument list
`va_end`	end varying-length argument list processing
`va_start`	begin varying length argument list processing.

These macros and the type **`va_list`** are defined in the header file **`<stdarg.h>`**. For more information on **`<stdarg.h>`**, see the function description for **`va_start`**.

General Utility Functions

The four utility functions are

`bsearch`	perform a binary search
`pdset`	packed decimal conversion: double to packed decimal
`pdval`	packed decimal conversion: packed decimal to double
`qsort`	sort an array of elements.

Program Control Functions

The program entry mechanism, which is the means by which the **`main`** function gains control, is system dependent. However, program exit is not always system dependent, although it does have some implementation dependencies.

One simple way to terminate execution of a C program is for the **`main`** function to execute a **`return`** statement; another is for the **`main`** function to drop through its terminating brace. However, in many cases, a more flexible program exit capability is needed. This capability is provided by the **`exit`** function described in this section. This function offers the advantage of allowing any function (not just **`main`**) to terminate the program, and it allows information to be passed to other programs. *

You can use the **`atexit`** function to define a function to be called during normal program termination, either due to a call to **`exit`** or due to return from the **`main`** function.

The **`abend`** and **`abort`** functions can also be used to terminate execution of a C program. These functions cause abnormal termination, which causes both library cleanup and user cleanup (defined by **`atexit`** routines) to be bypassed.

In some cases, it is useful for a program to pass control directly to another part of the program (within a different function) without having to go through a long and possibly complicated series of function returns. The **`setjmp`** and **`longjmp`** functions

* For programs using the compiler **`indep`** feature, program execution can also be terminated by calling the L$UEXIT routine from non-C code, as described in Appendix 5, "Using the indep Option for Interlanguage Communication," in the *SAS/C Compiler and Library User's Guide*.

provide a general capability for passing control in this way.

You can use the SAS/C extension **blkjmp** to intercept calls to **longjmp** that cause the calling routine to be terminated. This is useful for functions that allocate resources that must be released before the function is terminated. You can also use **blkjmp** to intercept calls to **exit**.

Note: The jump functions use a special type, **jmp_buf**, which is defined in the **<setjmp.h>** header file.

Several of the program control functions have a special version for use in the Systems Programming Environment. See "Implementation of Functions" on page 1-15 for more details.

The program control functions are

abend	abnormally terminate execution via ABEND
abort	abnormally terminate execution
atexit	register program cleanup function
blkjmp	intercept nonlocal gotos
exit	terminate execution
longjmp	perform nonlocal goto
onjmp	define target for nonlocal goto
onjmpout	intercept nonlocal gotos
setjmp	define label for nonlocal goto.

Memory Allocation Functions

The standard library provides several different levels of memory allocation, providing varying levels of portability, efficiency, and convenience. The **malloc** family of functions (**calloc**, **malloc**, **free**, and **realloc**) conforms to the ISO/ANSI standard and is therefore the most portable (and usually the most convenient) technique for memory allocation. The **pool** family of functions (**pool**, **palloc**, **pfree**, and **pdel**) is not portable but is more efficient for many applications. Finally, the **sbrk** function provides compatibility with traditional UNIX low-level memory management but is inflexible because the maximum amount of memory that can be allocated is fixed independently of the size of the region or virtual machine. All of the memory allocation functions return a pointer of type **void *** that is guaranteed to be properly aligned to store any object.

All of these interfaces, except **sbrk**, use the operating system's standard memory allocation technique (GETMAIN under MVS, DMSFREE under CMS, or CMSSTOR under bimodal CMS) to allocate memory blocks. This means that blocks allocated by the C language may be interspersed with blocks allocated by the operating system or by other programs. It also means that the C program is always able to allocate memory up to the limits imposed by the region or virtual machine size.

If your application requires more complete control of memory allocation parameters, you can call the GETMAIN, DMSFREE, and CMSSTOR functions yourself, as described in Chapter 14, "Systems Programming with the SAS/C Compiler," of the *SAS/C Compiler and Library User's Guide.* Because the other memory allocation functions do not invoke the operating system every time they are called, they are generally more efficient than direct use of the operating system services.

Under MVS, all SAS/C memory allocation (except when the program invokes the GETMAIN SVC directly) is task related. Thus, it is not valid to use **malloc** to allocate a block of memory under one TCB and free it under another. Even if the two tasks share all MVS subpools, this error will cause memory management chains to become erroneously linked, which will eventually cause a memory management ABEND in one or more of the involved tasks.

Even the SAS/C pool allocation functions, such as **`palloc`**, do not allow memory allocation to be managed across task boundaries. **`palloc`** calls **`malloc`** to extend a pool if necessary; therefore, it may corrupt memory chains if used under the wrong task. Additionally, the code generated by **`palloc`** and **`pfree`** does no synchronization, which means that simultaneous use on the same pool in several tasks could cause the same element to be allocated twice, or lost from the memory chains.

If an application requires multiple SAS/C subtasks with memory shared between subtasks, we recommend that you assign to a single task the job of performing all shared memory allocation and deallocation for the application. All other tasks should then use POST/WAIT logic to request the allocation task to allocate or free memory. This design ensures that all shared memory is managed as a unit and avoids synchronization issues caused by simultaneous allocation requests.

The memory allocation functions are

`calloc`	allocate and clear memory
`free`	free a block of memory
`malloc`	allocate memory
`palloc`	allocate an element from a storage pool
`pdel`	delete a storage pool
`pfree`	return an allocated element to a storage pool
`pool`	allocate a storage pool
`realloc`	change the size of an allocated memory block
`sbrk`	traditional UNIX low-level memory allocation.

Diagnostic Control Functions

The functions in this category allow you to control the processing of errors by the library. For example, you can put diagnostics into programs with **`assert`**, generate a library traceback with **`btrace`**, and suppress library diagnostics with **`quiet`**.

The diagnostic control functions are

`assert`	put diagnostics into programs
`btrace`	generate a traceback
`perror`	write diagnostic message
`quiet`	control library diagnostic output
`strerror`	map error number to a message string.

Timing Functions

The SAS/C library supports all of the ISO/ANSI timing functions. Timing functions allow determination of the current time of day, and the processing and formatting of time values. Programs using any of these functions must include the header file **`<time.h>`**.

The POSIX standards mandate several changes to the SAS/C timing functions. As a result, the SAS/C Release 6.00 library assumes a new default epoch and can process time-zone information defined via the **`TZ`** environment variable.

In previous releases of SAS/C, **`time_t`** values were measured from the 370 epoch, starting at January 1, 1900. In accordance with the POSIX specification , the SAS/C Release 6.00 library measures time from the UNIX epoch, starting at January 1, 1970.

A program with special requirements can specifically define its own epoch by declaring the extern variable **_epoch**, as in the following example:

```
#include <time.h>

time_t _epoch = _EPOCH_370;
```

This declaration specifies the 370 epoch. You can also use the value **_EPOCH_UNIX** to specify the standard UNIX epoch. Any legitimate **time_t** value can be used as the epoch, as in this example, which defines the start of the epoch as January 1, 1971:

```
#include <time.h>

time_t _epoch = _EPOCH_UNIX+365*86400;
```

Also, if the **TZ** environment variable is set, the SAS/C **mktime**, **ctime**, **localtime**, and **strftime** routines will take time-zone information into account. For TSO or CMS programs, **TZ** may be defined as an external or permanent scope environment variable.

Note: The **TZ** environment variable expresses offset from Greenwich mean time. The SAS/C library assumes that the hardware time-of-day clock has been set to accurately reflect Greenwich time, as recommended by the IBM ESA Principles of Operation. If the hardware time-of-day clock does not accurately reflect Greenwich time, then processing of the **TZ** information will not be correct, and applications depending on accurate local time information may fail.

The **<time.h>** header file defines two types that describe time values: **time_t** and **struct tm**. The type **time_t** is a numeric type used to contain time values expressed in the seconds after some implementation-defined base point (or era). The type **struct tm** is a structure that is produced by several of the timing routines; it contains time and date information in a more readily usable form. The **struct tm** structure is defined to contain the following components:

```
int tm_sec;     /* seconds after the minute (0-59)  */
int tm_min;     /* minutes after the hour (0-59)    */
int tm_hour;    /* hours since midnight (0-23)      */
int tm_mday;    /* day of the month (1-31)          */
int tm_mon;     /* months since January (0-11)      */
int tm_year;    /* years since 1900                 */
int tm_wday;    /* days since Sunday (0-6)          */
int tm_yday;    /* days since January 1 (0-365)     */
int tm_isdst;   /* Daylight Savings Time flag.      */
```

Routines are provided to convert **time_t** values to **struct tm** values and to convert either of these types to a formatted string suitable for printing.

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.

The timing functions are

asctime	convert time structure to character string
clock	measure program processor time

`ctime`	convert local time value to character string
`difftime`	compute the difference of two times
`gmtime`	break Greenwich mean time into components
`localtime`	break local time value into components
`mktime`	generate encoded time value
`strftime`	convert time to string
`time`	return the current time
`tzset`	store time zone information.

I/O Functions

The SAS/C library provides a large set of input/output functions. These functions are divided into two groups, standard I/O functions and UNIX style I/O functions.

The library's I/O implementation is designed to

- support the ISO/ANSI C standard
- support the execution of existing programs developed with other C implementations
- support the development of new portable programs
- support the effective use of native MVS and CMS I/O facilities and file types.

The library provides several I/O techniques to meet the needs of different applications. To achieve the best results, you must make an informed choice about the techniques to use. Criteria that should influence your choice are

- the need for portability (For example, will the program execute on several different systems?)
- the required capabilities (For example, will the program need to alter the file position randomly during processing?)
- the need for efficiency (For example, can the program accept some restrictions on file format to achieve good performance?)
- the intended use of the files (For example, will files produced by the program later be processed by an editor or by a program written in another language?).

To make these choices, you need to understand general C I/O concepts as well as the native I/O types and file structures supported by the 370 operating systems, MVS and CMS.

Details about C I/O concepts and functions can be found in Chapter 3, "I/O Functions" on page 3-1.

The I/O functions are

`afflush`	flush file buffers to disk
`afopen`	open a file with system-dependent options
`afread`	read a record
`afread0`	read a record (possibly length zero)
`afreadh`	read part of a record
`afreopen`	reopen a file with system-dependent options
`afwrite`	write a record
`afwrite0`	write a record (possibly length 0)
`afwriteh`	write part of a record
`aopen`	open a file for UNIX style access with amparms

`clearerr`	clear error flag
`close`	close a file opened by **`open`**
`_close`	close an HFS file
`closedir`	close a directory
`clrerr`	clear error flag and return status
`creat`	create and open a file for UNIX style I/O
`ctermid`	get a filename for the terminal
`dup`	duplicate a file descriptor
`dup2`	duplicate a file descriptor to a specific file descriptor number
`fattr`	return file attribute information
`fclose`	close a file
`fcntl`	control open files or sockets
`_fcntl`	control open file descriptors for OpenEdition HFS files
`fdopen`	access an OpenEdition file descriptor via standard I/O
`feof`	test for end of file
`ferror`	test error flag
`ffixed`	test for fixed-length records
`fflush`	flush output buffer
`fgetc`	read a character from a file
`fgetpos`	store the current file position
`fgets`	read a string from a file
`fileno`	return file descriptor number
`fnm`	return filename
`fopen`	open a file
`fprintf`	write formatted output to a file
`fputc`	write a character to a file
`fputs`	write a string to a file
`fread`	read items from a file
`freopen`	reopen a file
`fscanf`	read formatted input from a file
`fseek`	reposition a file
`fsetpos`	reposition a file
`fsync`	flush buffers for a UNIX style file to disk
`_fsync`	flush HFS file buffers to disk
`ftell`	obtain the current file position
`fterm`	terminal file test
`ftruncate`	truncate an OpenEdition file
`fwrite`	write items to a file
`getc`	read a character from a file

getchar	read a character from the standard input stream
gets	read a string from the standard input stream
isatty	test for terminal file
kdelete	delete current record from keyed file
kgetpos	return position information for keyed file
kinsert	insert record into keyed file
kreplace	replace record in keyed file
kretrv	retrieve next record from keyed file
ksearch	search keyed file for matching record
kseek	reposition a keyed stream
ktell	return RBA of current record
lseek	position a file opened for UNIX style access
_lseek	position an OpenEdition HFS file
open	open a file for UNIX style I/O
_open	open an OpenEdition HFS file
opendir	open an OpenEdition HFS directory
pclose	close a pipe opened by **popen**
pipe	create and open a pipe
popen	open pipe I/O to an OpenEdition shell command
printf	write formatted output to the standard output stream
putc	write a character to a file
putchar	write a character to the standard output stream
puts	write a string to the standard output stream
read	read data from a file opened for UNIX style access
_read	read data from an OpenEdition HFS file
readdir	read an OpenEdition directory entry
rewind	position to start of file
rewinddir	positions an OpenEdition directory stream to the beginning
scanf	read formatted data from the standard input stream
setbuf	change stream buffering
setvbuf	change stream buffering
snprintf	write a limited amount of formatted output to a string
sprintf	write formatted output to a string
sscanf	read formatted data from a string
tmpfile	create and open a temporary file
tmpnam	generate temporary filename
ttyname	get name of open terminal file
ungetc	push back an input character
vfprintf	write formatted output to a file
vprintf	write formatted output to the standard output stream

`vsnprintf`	write a limited amount of formatted output to a string
`vsprintf`	write formatted output to a string
`write`	write data to a file open for UNIX-style access
`_write`	write data to an OpenEdition HFS file.

File Management Functions

The SAS/C library provides a number of file management functions that enable you to interact with the file system in various ways. For example, functions are provided to test and change file attributes, remove or rename files, and search for all files whose names match a pattern. The file management functions are

`access`	test for the existence and access privileges of a file
`_access`	test for OpenEdition HFS file existence or access privileges
`chdir`	change the OpenEdition working directory
`chmod`	change the protection bits of an OpenEdition HFS file
`cmsdfind`	find the first CMS fileid matching a pattern
`cmsdnext`	find the next CMS fileid matching a pattern
`cmsffind`	find the first CMS fileid matching a pattern
`cmsfnext`	find the next CMS fileid matching a pattern
`cmsfquit`	release data held by `cmsffind`
`cmsstat`	fill in a structure with information about a CMS file
`fchmod`	change the protection bits of an OpenEdition file
`fstat`	get status information for an OpenEdition file
`getcwd`	return the name of the OpenEdition working directory
`link`	create a hard link to an existing OpenEdition file
`lstat`	get status information about an OpenEdition file or symbolic link
`mkdir`	create a new OpenEdition directory
`mkfifo`	create an OpenEdition FIFO special file
`oeddinfo`	get information about a DD statement allocated to an OpenEdition HFS file
`osddinfo`	obtain information about a data set by DDname
`osdfind`	find the first MVS file/member matching a pattern
`osdnext`	find the next MVS file/member matching a pattern
`osdquit`	terminate MVS file/member search
`osdsinfo`	obtain information about an MVS data set by dsname
`readlink`	read the contents of an OpenEdition symbolic link
`remove`	delete a file
`rename`	rename a file
`_rename`	rename an OpenEdition disk file or directory
`rmdir`	remove an empty OpenEdition directory
`sfsstat`	return information about a CMS shared file system file or directory
`stat`	get status information for an OpenEdition file

`symlink`	create a symbolic link to an OpenEdition file
`unlink`	delete a file
`_unlink`	delete an OpenEdition HFS file
`utime`	update the access and modification times for an OpenEdition file.

System Interface and Environment Variables

The system interface and environment variables enable a program to interact with the operating system. These functions are described in detail in Chapter 4, "Environment Variables" on page 4-1.

The system interface functions are

`clearenv`	delete environment variables
`cuserid`	get current userid
`getenv`	get value of environment variable
`getlogin`	determine user login name
`iscics`	return CICS environment information
`oslink`	call an MVS utility program
`putenv`	update environment variable
`setenv`	update environment variable
`system`	execute a system command.

Signal-Handling Functions

The signal-handling feature of the SAS/C library is a collection of library functions that enables you to handle unexpected conditions and interrupts during execution of a C program. These functions are described in detail in Chapter 5, "Signal-Handling Functions" on page 5-1. Using this facility, you can

- define which signals are managed by the SAS/C library and which ones are managed by OpenEdition
- define a function called a signal handler that performs special processing when a signal occurs
- control which signals are processed by your program, which are ignored, and which are handled using the system default action
- block the reception of some signals
- generate signals
- define signals for your own needs.

The signal-handling functions are

`alarm, alarmd`	request a signal after a real-time interval
`ecbpause`	delay program execution until the occurrence of a C signal or the POSTing of an Event Control Block (obsolete function)
`ecbsuspend`	delay program execution until the occurrence of a C signal or the POSTing of an Event Control Block
`kill`	send a signal to a process
`oesigsetup`	define which signals are managed by the SAS/C library and which signals are managed by OpenEdition
`pause`	suspend execution until a signal is received

`raise`	generate an artificial signal
`sigaction`	define a signal handler
`sigaddset` **`sigdelset`** **`sigemptyset`** **`sigfillset`** **`sigismember`**	manipulate **`sigset_t`** objects
`sigblock`	inhibit discovery of asynchronous signals (obsolete function)
`sigchk`	check for asynchronous signals
`siggen`	generate an artificial signal with additional information
`siginfo`	obtain information about a signal
`siglongjmp`	restore a previously saved stack environment and signal mask
`signal`	define program signal handling
`sigpause`	suspend execution and block discovery of signals (obsolete function)
`sigpending`	determine pending signals for a process
`sigprocmask`	inhibit or permit discovery of signals
`sigsetjmp`	saves the current stack environment and signal mask
`sigsetmask`	inhibit or permit discovery of signals (obsolete function)
`sigsuspend`	suspend program execution until a signal is generated
`sleep, sleepd`	suspend execution for a period of time.

3 I/O Functions

Introduction

The SAS/C library provides a large set of input/output functions, which are divided into two groups, standard-style I/O functions and UNIX style I/O functions. This chapter describes these functions and how they are used.

The following section describes how to perform input and output using the functions provided in the SAS/C library. This section is important if you use SAS/C I/O facilities, whether you are developing new programs or porting existing programs from other environments.*

The library's I/O implementation is designed to

- ☐ support the ISO/ANSI C standard
- ☐ support the execution of existing programs developed with other C implementations
- ☐ support the development of new portable programs
- ☐ support the effective use of native MVS and CMS I/O facilities and file types.

* In addition to the traditional C I/O facilities described in this section, the library offers for both CMS and MVS a set of functions to perform low-level I/O, making direct use of native I/O facilities. These facilities are described in Chapter 2, "CMS Low-Level I/O Functions," and Chapter 3, "MVS Low-Level I/O Functions," in *SAS/C Library Reference, Volume 2.*

As described later in this chapter, the library provides several I/O techniques to meet the needs of different applications. To achieve the best results, you must make an informed choice about the techniques to use. Criteria that should influence this choice are

- the need for portability (For instance, will the program execute on several different systems?)
- the required capabilities (For instance, will the program need to alter the file position randomly during processing?)
- the need for efficiency (For instance, can the program accept some restrictions on file format to achieve good performance?)
- the intended use of the files (For instance, will files produced by the program later be processed by an editor or by a program written in another language?).

To make these choices, you need to understand general C I/O concepts as well as the native I/O types and file structures supported by the 370 operating systems, MVS and CMS. These topics are addressed in this chapter. The description is aimed primarily at the knowledgeable C programmer who should be familiar with 370 I/O concepts. In many cases, understanding the 370 I/O concepts is necessary to control and anticipate program behavior. Where possible, this chapter addresses these issues, but familiarizing yourself with 370 I/O concepts using other sources is highly recommended. Chapter 1, "Introduction," of the *SAS/C Compiler and Library User's Guide* lists the documents from International Business Machines Corporation that may be of particular value.

Some parts of this chapter are intended for knowledgeable 370 programmers who may be interested in the relationship between SAS/C I/O and traditional 370 I/O techniques. These portions are identified as such, and you can skip them if you do not have the necessary background in 370 I/O concepts.

This chapter is divided into two sections: technical background and technical summaries. For the most effective use of SAS/C I/O techniques, you should become familiar with the concepts presented in the next section, "Technical Background." Skim "Technical Summaries" on page 3-22 for information relevant to your application, and consult specific I/O function descriptions for details on the functions. Much of the material in the last two sections is reference information of limited applicability, but understanding the technical background section is essential for effective use of the library I/O functions.

Technical Background

This section provides a fairly in-depth summary of the fundamentals of C I/O. It begins with a discussion of traditional C I/O concepts, then discusses UNIX low-level, ISO/ANSI, and IBM 370 I/O concepts. These concepts are combined in "SAS/C I/O Concepts" on page 3-14 and "370 Perspectives on SAS/C Library I/O" on page 3-18. The final section provides guidelines for choosing an I/O method, based on the needs of your application.

Traditional C (UNIX) I/O Concepts

When C was initially designed, no library, and therefore no I/O, was included. It was assumed that libraries suitable for use with particular systems would be developed. Because most early use of the C language was associated with UNIX operating systems, the UNIX I/O functions were considered the standard I/O method for C. As the C language has evolved, the I/O definition has changed to some extent, but understanding the underlying UNIX concepts is still important.

In addition, many useful C programs were first developed under UNIX operating systems, and such programs frequently are unaware of the existence of other systems or I/O techniques. Such programs cannot run on systems as different from UNIX as CMS or MVS without carefully considering their original environment.

The UNIX I/O model

The main features of the UNIX I/O model are as follows:

- A file is a sequence of characters. A file contains no information other than these characters. It is possible to create a file containing no characters.
- A file is divided into *lines* by the new-line character ('\n'). New-line characters have no other special properties. A file may contain lines of any length, including 0.
- The characters in a file are numbered sequentially, starting at 0. It is possible to position a file efficiently at any particular character.
- No arbitrary restrictions are imposed on the lengths of lines in a file or on the size of a file. Padding characters are never written to fill out a file or a line to a particular length or boundary.
- Files can be opened for reading, writing, or both. When a file is opened for writing, the previous contents can optionally be erased. After the file is opened, characters can be replaced but not removed. That is, the end-of-file position can be advanced but not moved backwards.

UNIX Low-Level I/O

One complication in programs developed under UNIX operating systems is that UNIX defines two different I/O interfaces: standard I/O and low-level I/O (sometimes called unbuffered I/O). *Standard I/O* is a more portable form of I/O than low-level I/O, and UNIX documentation recommends that portable programs be written using this form. However, UNIX *low-level I/O* is widely recognized as more efficient than standard I/O, and it provides some additional capabilities, such as the ability to test whether a file exists before it is opened. For these and other reasons, many programs use low-level I/O, despite its documented lack of portability.

UNIX operating systems also support a *mixed-level* form of I/O, wherein a file is accessed simultaneously with standard I/O and low-level I/O. C implementations that support the UNIX low-level functions may be unable to support mixed-level I/O, if the two forms of I/O are not closely related in the UNIX manner.

UNIX low-level I/O is not included in the ISO/ANSI C standard, so it may be unavailable with recently developed C compilers. Also, do not assume that this form of I/O is truly low-level on any system other than UNIX.

ISO/ANSI C I/O Concepts

The definition of the C I/O library contained in the ISO/ANSI C standard is based on the traditional UNIX standard I/O definition, but differs from it in many ways. These differences exist to support efficient I/O implementations on systems other than UNIX, and to provide some functionality not offered by UNIX. In general, where definitions of I/O routines differ between ISO/ANSI C and UNIX C, programs should assume the ISO/ANSI definitions for maximum portability. The ISO/ANSI definitions are designed for use on many systems including UNIX, while the applicability of the UNIX definitions is more limited.

Text access and binary access

In the UNIX I/O model, files are divided into lines by the new-line character ('\n'). For this reason, C programs that process input files one line at a time traditionally read characters until a new-line character is encountered. Similarly, programs that write output one line at a time write a new-line character after each line of data.

Many systems other than UNIX use other conventions for separating lines of text. For instance, the IBM PC operating system, PC DOS, separates lines of text with two characters, a carriage return followed by a line feed. The IBM 370 uses yet another method. To enable a line-oriented C program written for UNIX to execute under PC DOS, a C implementation must translate a carriage return and line feed to a new-line character on input, and must translate a new-line character to a carriage return and line feed on output. Although this translation is appropriate for a line-oriented program, it is not appropriate for other programs. For instance, a program that writes object code to a file cannot tolerate replacement of a new-line character in its output by a carriage return and a line feed. For this reason, most systems other than UNIX require two distinct forms of file access: text access and binary access.

The ISO/ANSI I/O definition requires that when a program opens a file, it must specify whether the file is to be accessed as a text stream or a binary stream. When a file is accessed as a *binary stream*, the implementation must read or write the characters without modification. When a file is accessed as a *text stream*, the implementation must present the file to the program as a series of lines separated by new-line characters, even if a new-line character is not used by the system as a physical line separator. Thus, under PC DOS, when a program writes a file using a binary stream, any new-line characters in the output data are written to the output file without modification. But when a program writes a file using a text stream, a new-line character in the output data is replaced by a carriage return and a line feed to serve as a standard PC DOS line separator.

If a file contains a real new-line character (one that is not a line separator) and the file is read as a text stream, the program will probably misinterpret the new-line character as a line separator. Similarly, a program that writes a carriage return to a text stream may generate a line separator unintentionally. For this reason, the ISO/ANSI library definition leaves the results undefined when any nonprintable characters (other than horizontal tab, vertical tab, form feed, and the new-line character) are read from or written to a text stream. Therefore, text access should be used only for files that truly contain text, that is, lines of printable data.

Programs that open a file without explicitly specifying binary access are assumed to require text access, because the formats of binary data, such as object code, vary widely from system to system. Thus, portable programs are more likely to require text access than binary access.

Padding

Many non UNIX file systems require files to consist of one or more data blocks of a fixed size. In these systems, the number of characters stored in a file must be a multiple of this block size. This requirement can present problems for programs that need to read or write arbitrary amounts of data unrelated to the block size; however, it is not a problem for text streams. When a text stream is used, the implementation can use a control character to indicate the logical end of file. This approach cannot be used with a binary stream, because the implementation must pass all data in the file to the program, whether it has control characters or not.

The ISO/ANSI C library definition deals with fixed data blocks by permitting output files accessed as binary streams to be padded with null ('\0') characters. This padding permits systems that use fixed-size data blocks to always write blocks of the correct size. Because of the possibility of padding, files created with binary streams on such systems may contain one or more null characters after the last character written by the program. Programs that use binary streams and require an exact end-of-file indication must write their own end-of-file marker (which may be a control character or sequence of control characters) to be portable.

A similar padding concern can occur with text access. Some systems support files where all lines must be the same length. (Files defined under MVS or CMS with record format F are of this sort.) ISO/ANSI permits the implementation to pad output lines with blanks when these files are written and to remove the blanks at the end of lines when the files are read. (A blank is used in place of a null character, because text access requires a printable padding character.) Therefore, portable programs write lines containing trailing blanks and expect to read the blanks back if the file will be processed later as input.

Similarly, some systems (such as CMS) support only nonempty lines. Again, ISO/ANSI permits padding to circumvent such system limitations. When a text stream is written, the Standard permits the implementation to write a line containing a single blank, rather than one containing no characters, provided that this line is always read back as one containing no characters. Therefore, portable programs should distinguish empty lines from ones that contain a single blank.

Finally, some systems (such as CMS) do not permit files containing no characters. A program is nonportable if it assumes a file can be created merely by opening it and closing it, without writing any characters.

File positioning with fseek and ftell

As stated earlier, the UNIX I/O definition features seeking by character number. For instance, it is possible to position directly to the 10,000th character of a file. On a system where text access and binary access are different, the meaning of a request to seek to the 10,000th character of a text stream is not well defined. **`ftell`** and **`fseek`** enable you to obtain the current file position and return to that position, no matter how the system implements text and binary access.

Consider a system such as PC DOS, where the combination of carriage return and line feed is used as a line separator. Because of the translation, a program that counts the characters it reads is likely to determine a different character position from the position maintained by the operating system. (A line that the program interprets as n characters, including a final new-line character, is known by the operating system to contain $n + 1$ characters.)

Some systems, such as the 370 operating systems, do not record physical characters to indicate line breaks. Consider a file on such a system composed of two lines of data, the first containing the single character `1` and the second containing the single character `2`. A program accessing this file as a text stream receives the characters `1\n2\n`. The program must process four characters, although only two are physically present in the file. A request to position to the second character is ambiguous. The library cannot determine whether the next character read should be `\n` or `2`.

Even if you resolve the ambiguity of file positioning in favor of portability (by counting the characters seen by the program rather than physical characters), implementation difficulties may preclude seeking to characters by number using a text stream. Under PC DOS, the only way to seek accurately to the 10,000th character of a file is to read 10,000 characters because the number of carriage return and line feed pairs in the file is not known in advance. If the file is opened for both reading and writing, replacing a printable character with a new-line character requires replacing one physical character with two. This replacement requires rewriting the entire file after the point of change. Such difficulties make it impractical on many systems to seek for text streams based on a character number.

Situations such as those discussed in this section show that on most systems where text and binary access are not identical, positioning in a text stream by character number cannot be implemented easily. Therefore, the ISO/ANSI standard permits a library to implement random access to a text stream using some indicator of file position other than character number. For instance, a file position may be defined as a value derived from the line number and the offset of the character in the line.

File positions in text streams cannot be used arithmetically. For instance, you cannot assume that adding 1 to the position of a particular character results in the position of the next character. Such file positions can be used only as tokens. This means that you can obtain the current file position (using the **`ftell`** function) and later return to that position (using the **`fseek`** function), but no other portable use of the file position is possible.

This change from UNIX behavior applies only to text streams. When you use **`fseek`** and **`ftell`** with a binary stream, the ISO/ANSI standard still requires that the file position be the physical character number.

File positioning with fgetpos and fsetpos

Even with the liberal definition of random access to a text stream given in the previous section, implementation of random access can present major problems for a file system that is very different from that of a traditional UNIX file system. The traditional MVS file system is an example of such a system. To assist users of these file systems, the Standard includes two non UNIX functions, `fsetpos` and `fgetpos`.

File systems like the MVS file system have two difficulties implementing random access in the UNIX (ISO/ANSI binary) fashion:

- They do not record character-oriented position information. For many MVS files, such as those with record format VB, a request to position to the 10,000th character can be satisfied only by positioning to the first character and then reading until 10,000 characters have been read. (To determine the number of characters in the file, it is necessary to read the entire file.)
- Some files may contain more characters than the largest possible `long int` value. Because UNIX operating systems and the Standard define the file position to have type `long int`, random access to all such enormous files cannot be supported. The functions `fgetpos` and `fsetpos` are defined by the Standard to perform operations similar to those of `fseek` and `ftell`, except that the representation of a file position is completely implementation-defined. This allows an implementation to choose a representation for the file position that is large enough to address all characters of the largest possible file and that can take into account all the idiosyncrasies of the host operating system. (For example, the file position may reference a disk track number rather than a record number or byte number.) Thus, using `fgetpos` and `fsetpos` for random access produces the greatest likelihood that a program will run on a system dissimilar to UNIX.

The `fsetpos` and `fgetpos` functions did not exist prior to the definition of the ISO/ANSI C standard. Because many C libraries have not yet implemented them, they are at this time less portable than `fseek` and `ftell`, which are compatible with UNIX operating systems.

However, it is a relatively straightforward task to implement them as macros that call `fseek` and `ftell` in such systems. After these macros have been written, `fsetpos` and `fgetpos` are essentially as portable as their UNIX counterparts and will offer substantial additional functionality where provided by the library on systems such as MVS.

The ISO/ANSI I/O model

The following list describes the I/O model for ISO/ANSI C. The points are listed in the same order as the corresponding points for the UNIX I/O model, as presented in the previous section.

- A file may be processed in one of two ways: as a text stream, or as a binary stream. When a file is processed as a binary stream, it appears to the program as a sequence of characters. It may not be possible to create a file containing no characters.
- A file accessed as a text stream appears to the program as a sequence of lines separated by occurrences of the new-line character (`'\n'`). The effects of reading or writing control characters using a text stream are not predictable. An implementation is permitted to record line separators using some technique other than physical new-line characters.

- When a file is accessed as a binary stream, its characters are numbered sequentially starting at 0. It is possible to position a binary stream to any particular character. When a file is accessed as a text stream, its characters are addressable, but not necessarily by a physical character number. It is possible to position a file accessed as a text stream to any character, provided the address of that character was obtained at the time of some previous access to that character.
- An implementation may restrict the size of lines or files. The implementation may pad files accessed as a binary stream with null characters at the end of the file, and it may pad files accessed as a text stream with blanks at the end of each line.
- Files can be opened for reading or writing, or both. When a file is opened for writing, the previous contents can optionally be erased. It is undefined whether writing a character before the end of file shortens the length of the file or leaves it unchanged.

IBM 370 I/O Concepts

Programmers accustomed to other systems frequently find the unique nature of 370 I/O confusing. This section organizes the most significant information about 370 I/O for SAS/C users. Note that this description is general rather than specific. Details and complex special cases are generally omitted to avoid obscuring the basic principles. See the introduction to the *SAS/C Compiler and Library User's Guide* for a small bibliography of relevant IBM publications that should be consulted for additional information.

Fundamental principles

- There are two 370 operating systems of interest, MVS and CMS. They implement different file systems. (CMS also implements *OS simulation*, which emulates MVS I/O under CMS. The emulation is not perfect and is actually a third I/O implementation.)
- Many file systems feature exactly one kind of file. For instance, in UNIX all files are simply sequences of characters. The 370 operating systems, especially MVS, go to the opposite extreme and handle many different types of files, each with its own peculiarities and uses. In general, the programmer must decide during program design which sorts of files a program will use.
- 370 I/O is record oriented. That is, files are treated as sequences of records, not sequences of characters. The idea that a physical character or character sequence may be used as a record or line separator is completely alien to the 370 systems. (An analogy that may be helpful is that UNIX operating systems and PC DOS treat line-oriented files as virtual terminals, with lines separated by layout characters such as the new-line and form feed characters. The 370 systems handle files as if they were virtual card decks consisting of physical records separated by gaps.)
- Most file systems allow the same program to replace old data in a file and to add new data at the end. In general, 370 I/O does not permit you to mix these two kinds of updates within the same program. When a file is opened using a technique that permits the addition of new data, the replacement of old data generally causes any following data to be discarded.
- 370 I/O is hardware oriented. It uses physical disk addresses to encode file positions. Under MVS, you cannot address records efficiently, even with a record number. For common file types, you must use an actual disk address to position to a record without reading from the start of a file.
- Another aspect of the hardware orientation of 370 I/O is the large number of file attributes that must be assigned, either by the program or by the user. Many of these attributes have no effect other than to alter the physical layout of the data. Such attributes are defined for the sole purpose of enabling the programmer to trade off various aspects of program performance. For example, you can permit a program to execute faster by using more memory for buffer space. In some cases, the ability to

tailor these attributes is vital, but frequently the programmer is forced to make such choices when performance is not an important consideration.

- □ The 370 file systems are lacking in disk space management. This means that programs must deal with the inability to enlarge files. It also means that users must provide size estimates to the system when files are created. It is necessary with some commonly used file types to run utilities to reclaim wasted file space. These problems are most notable under MVS, but they can also be a factor under CMS.
- □ For programmers accustomed to the UNIX file system, the conventions for 370 file naming may seem strange. Under MVS, filenames are often given only as indirect names (DDnames in MVS jargon) that can be connected to actual filenames only by the use of a control language. (It is possible to refer to a file by its actual name rather than a DDname, but the absence of directories and reliable user identification under MVS make this an inconvenient and often difficult technique.) Under CMS, either DDnames or more natural filenames can be used, but some programs choose to use DDnames to achieve closer compatibility with MVS.

File organizations under MVS

Under MVS, files are classified first by file organization. A number of different organizations are defined, each tailored for an expected type of usage. For instance, files with sequential organization are oriented towards processing records in sequential order, while most files with VSAM (Virtual Storage Access Method) organization are oriented toward processing based on key fields in the data.

For each file organization, there is a corresponding MVS access method for processing such files. (An *MVS access method* is a collection of routines that can be called by a program to perform I/O.) For instance, files with sequential organization are normally processed with the Basic Sequential Access Method (BSAM). Sometimes, a file can be processed in more than one way. For example, files with direct organization can be processed either with BSAM or with the Basic Direct Access Method (BDAM).

The file organizations of most interest to C programmers are *sequential* and *partitioned.* The remainder of this section relates primarily to these file organizations, but many of the considerations apply equally to the others. A number of additional considerations apply specifically to files with partitioned organization. These considerations are summarized in “MVS partitioned data sets” on page 3-10.

Note: An important type of MVS file, the Virtual Storage Access Method (VSAM) file, was omitted from the previous list. VSAM files are organized as records identified by a character string or a binary key. Because these files differ so greatly from the expected C file organization, they are difficult to access using standard C functions. Because of the importance of VSAM files in the MVS environment, full access to them is provided by nonportable extensions to the standard C library.

Note: Also, if your system supports OpenEdition MVS, it provides a hierarchical file system similar to the system offered on UNIX. The behavior of files in the hierarchical file system is described in “UNIX Low-Level I/O” on page 3-3. Only traditional MVS file behavior is described here.

The characteristics of a sequential or partitioned file are defined by a set of attributes called data control block (DCB) parameters. The three DCB parameters of most interest are record format (RECFM), logical record length (LRECL), and block size (BLKSIZE).

As stated earlier, MVS files are stored as a sequence of records. To improve I/O performance, records are usually combined into blocks before they are written to a device. The record format of a file describes how record lengths are allowed to vary and how records are combined into blocks. The logical record length of a file is the maximum length of any record in a file, possibly including control information. The block size of a file is the maximum size of a block of data.

The three primary record formats for files are F (fixed), V (variable), and U (undefined). Files with record format F contain records that are all of equal length. Files with format V or U may contain records of different lengths. (The differences between V and U are mostly technical.) Files of both F and V format are frequently used; the preferred format for specific kinds of data (for instance, program source) varies from site to site.

Ideally, the DCB parameters for a file are not relevant to the C program that processes it, but sometimes a C program has to vary its processing based on the format of a file, or to require a file to have a particular format. Some of the reasons for this are as follows:

- Because most C programs do not write lines of equal length, a C library implementation must add trailing blanks to the end of output lines in a record format F file and remove them on input. If this is inappropriate for an application, you may need to require the use of a record format V or U file, or to use a nonportable function to inhibit library padding.
- When writing to a file with a small logical record length as a text stream, the library may be forced to divide a long line into several records. In this case, when the file is read, the data are not identical to what was written.
- Some programs and system utilities require specific DCB attributes. For instance, the MVS linkage editor cannot handle object files whose block size is greater than 3200 bytes. C programs producing input for such programs must be aware of these requirements.
- One of the secondary DCB attributes a file can have is the ANSI control characters (RECFM=A) option, which means that the first character position of each record will be used as a FORTRAN carriage control character. The UNIX convention of using characters such as form feed and carriage return to create page formatting can be used only when the output file is defined to use ANSI control characters. Since some editors do not allow such files to be edited, it is generally not appropriate to assign this attribute to all files.
- The standard C language does not provide any way for you to interrogate or define file attributes. In cases in which a program depends on file attribute information, you have two choices. You can use control language when files are created or used to define the file attributes, or you can use nonportable mechanisms to access or specify this information during execution.

File organizations under CMS

Like most operating systems, CMS has its own native file system. (In fact, it has two: the traditional minidisk file system and the more hierarchical shared file system.) Unlike most operating systems, CMS has the ability to simulate the file systems of other IBM operating systems, notably OS and VSE. Also, CMS can transfer data between users in spool files with the VM control program (CP).

Therefore, CMS files are classified first by the type of I/O simulation (or lack thereof) used to read or write to them. The three types are

- CMS-format files, which are read and written by native CMS I/O support. This category includes spool files (virtual reader and printer files) and CMS disk files, either mini-disk based or in the shared file system.
- OS-format files, particularly MACLIBs and TXTLIBs (simulated OS PDS’s) and OS files on OS disks. These files are read and written by the CMS simulation of OS BSAM and other OS access methods.
- VSE-format files, particularly VSAM files, including VSAM files on OS or VSE disks. These files are read and written by the VSE implementation of VSAM under CMS.

CMS I/O simulation can be used to read files created by OS or VSE, but these operating systems cannot read files created by CMS, even when the files are created using CMS's simulation of their I/O system. In general, CMS adequately simulates OS and VSE file organizations, and the rules that apply in the *real* operating system also apply under CMS. However, the simulation is not exact. CMS's simulation differs in some details and some facilities are not supported at all.

CMS-format files, particularly disk files, are of most interest to C programmers. CMS disk files have a logical record length (LRECL) and a record format (RECFM). The LRECL is the length of the largest record; it may vary between 1 and 65,535. The RECFM may be F (for files with fixed-length records) or V (for files with variable-length records). Other file attributes are handled transparently under CMS. Files are grouped by *minidisk*, a logical representation of a physical direct-access device. The attributes of the minidisk, such as writability and block size, apply to the files it contains. Files in the shared file system are organized into directories, conceptually similar to UNIX directories.

Records in RECFM F files must all have the same LRECL. The LRECL is assigned when the file is created and may not be changed. Some CMS commands require that input data be in a RECFM F file. To support RECFM F files, a C implementation must either pad or split lines of output data to conform to the LRECL, and remove the padding from input records.

RECFM V files have records of varying length. The LRECL is the length of the longest record in the file, so it may be changed at any time by appending a new record that is longer than any other record. However, changing the record length of RECFM V files causes any following records to be erased. The length of any given record can be determined only by reading the record. (Note that the CMS LRECL concept is different from the MVS concept for V format files, as the LRECL under MVS includes extra bytes used for control information.)

Some rules apply for both RECFM F and RECFM V files. Records in CMS files contain only data. No control information is embedded in the records. Records may be updated without causing loss of data. Files may be read sequentially or accessed randomly by record number.

As under MVS, files that are intended to be printed reserve the first character of each record for an ANSI carriage control character. Under CMS, these files can be given a filetype of LISTING, which is recognized and treated specially by commands such as PRINT. If a C program writes layout characters, such as form feeds or carriage returns, to a file to effect page formatting, the file should have the filetype LISTING to ensure proper interpretation by CMS.

Be aware that the standard C language does not provide any way for you to interrogate or define file attributes. In cases in which a program depends on file attribute information, you have two choices. You can use the FILEDEF command to define file attributes (if your program uses DDnames), or you can use nonportable mechanisms to access or specify this information during execution.

MVS partitioned data sets

As stated earlier, one of the important MVS file organizations is the partitioned organization. A file with partitioned organization is more commonly called a *partitioned data set* (PDS) or a *library.* A PDS is a collection of sequential files, called members, all of which share the same area of disk space. Each member has an eight-character member name. Under MVS, source and object modules are usually stored as PDS members. Also, almost any other sort of data may be stored as a PDS member rather than as an ordinary sequential file.

Partitioned data sets have several properties that make them particularly difficult for programs that were written for other file systems to handle:

- It is not possible to add data to the end of a PDS member. Because each member is usually adjacent to the end of the previous member on the disk, adding data to the end of one member would destroy the next one. To change the size of a PDS member, it usually is necessary to copy and rewrite the entire member.
- Members are always added to a PDS at the end of the file. For this reason, it is impossible to write to two members of the same PDS at the same time, as this causes the two members to overlap randomly.
- When a member is replaced in a PDS, the space used by any previous member with the same name is not reclaimed. This makes PDS's particularly susceptible to running out of space. It is necessary to run a system utility to reclaim unused space in a PDS.
- A member does not always occupy the same spot in a PDS. Because PDS file positions are represented relative to the start of the entire PDS, file positions may differ between identical copies of the same data, even if all file attributes are identical.

These limitations may cause ISO/ANSI-conforming programs to fail when they use PDS members as input or output files. For instance, it is reasonable for a program to assume that it can append data to the end of a file. But due to the nature of PDS members, it is not feasible for a C implementation to support this, except by saving a copy of the member and then replacing the member with the copy. Although this technique is viable, it is very inefficient in both time and disk space. (This tradeoff between poor performance and reduced functionality is one that must be faced frequently when using C I/O on the 370. PDS members, which are perhaps the most commonly used kind of MVS file, are the most prominent examples of such a tradeoff.)

Note: Recent versions of MVS support an extended form of PDS, called a PDSE. Some of the previously described restrictions on a PDS do not apply to a PDSE. For example, unused space is reclaimed automatically in a PDSE.

CMS MACLIBs and TXTLIBs

Two important OS-simulated file types on CMS are the files known as MACLIBs and TXTLIBs. Both of these are simulations of OS-partitioned data sets. MACLIBs are typically used to collect textual data or source code; TXTLIBs may contain only object code. Unlike OS PDS's, these files always have fixed-length, 80-character records.

In general, MACLIBs and TXTLIBs may not be written by OS-simulated I/O. Instead, data are added or removed a member at a time by CMS commands. Input from MACLIBs and TXTLIBs can be performed using either OS-simulation or native CMS I/O.

Identifying files

In UNIX operating systems and similar systems, files are identified in programs by name, and program flexibility with files is achieved by organizing files into directories. Files with the same name may appear in several directories, and the use of a command language to establish working directories enables the user of a program to define program input and output flexibly at run time.

In the traditional MVS file system, all files occupy a single name space. (This is an oversimplification, but a necessary one.) Programs that open files by a physical filename are limited to the use of exactly one file at a site. You can use several techniques to increase program flexibility in this area, none of which is completely satisfactory. These techniques include the following:

□ Specify filenames in TSO format. When the time-sharing option of MVS (TSO) is used, each user's files usually begin with a userid, thereby ensuring that the filenames chosen by different users do not overlap. By convention, a user running under TSO can omit the userid from a filename specification. This helps considerably for those programs that always run interactively and never in batch mode. However, userid is a TSO concept and, unless a site uses optional software (such as an IBM or other vendor security system), programs cannot be associated with a userid when running in batch.

□ Specify filenames as DDnames. Under MVS-batch, using DDnames to identify files is traditional. A DDname is an indirect name associated with an actual filename or device addressed by a DD statement in batch or an ALLOCATE command under TSO. Programs that use DDnames to identify files are completely flexible. They can produce printed output, terminal output, or disk output, depending only on their control language. Unfortunately, control language must always be used, because there are no default file definitions.

  Because most traditional filenames include periods, which are not permitted in DDnames, programs from other environments may need to be modified if they are to use DDnames, and if the logic of the program will withstand such a change.

□ Determine filenames dynamically at run time rather than putting them in the program. For instance, you may get filenames from the user or from a profile or configuration file. This is the most flexible technique, but it may require extensive program changes.

Under CMS, you can use other techniques to increase program flexibility:

□ The concept of the CMS minidisk replaces the UNIX directory concept. However, CMS minidisks are not arranged hierarchically, as UNIX directories are arranged. CMS minidisks are not identified by name or device address but by *filemode letter*, which is assigned by using the CMS ACCESS command and can be changed at any time. (Because the same filename may exist on several minidisks, it may be necessary to include a filemode letter in a filename to make it unambiguous.) In many ways, the minidisk with filemode letter A corresponds to the UNIX working directory, but this analogy is only approximate.

□ CMS filenames use spaces in filenames rather than periods. This is not a problem, because it is natural for a C library to treat the filename **`xyz.c`** as XYZ C under CMS.

□ The CMS shared file system is hierarchically arranged, so there is often a natural correspondence between a UNIX pathname and a shared filename. Unfortunately, the differing character conventions of CMS and UNIX will generally inhibit a UNIX oriented program from running unchanged with the shared file system. For example, the UNIX pathname **`/tools/asm/main.c`** is the same as the shared filename MAIN C TOOLS.ASM.

□ CMS supports using DDnames for filenames instead of physical filenames. This feature allows programs to be easily ported between MVS and CMS. The file referred to by a DDname must be defined by using the CMS FILEDEF command before a program that uses the DDname is executed.

File existence

Under MVS, the concept of file existence is not nearly so clear-cut as on other systems, due primarily to the use of DDnames and control language. Since DDnames are indirect filenames, the actual filename must be provided through control language before the start of program execution. If the file does not already exist at the time the DD statement or ALLOCATE command is processed, it is created at that time. Therefore, a file accessed with a DDname must already exist before program execution.

An alternate interpretation of file existence under MVS that avoids this problem is to declare that a file exists after a program has opened it for output. By this interpretation, a file created by control language immediately before execution does not yet exist. Unfortunately, this definition of existence cannot be implemented because of the following technicalities:

- □ MVS does not distinguish in the catalog or Volume Table of Contents (VTOC) between a newly created file that has never been written and one that has been written but is empty (contains no characters).
- □ Attempting to read a file that has never been written produces random results because MVS does not erase disk space when it is allocated or freed. This makes it impossible to distinguish an empty file from a newly created file by trying to read it.

A third interpretation of existence is to say that an MVS file exists if it contains any data (as recorded in the VTOC). This has the disadvantage of making it impossible to read an empty file but the much stronger advantage that a file created by control language immediately before program execution is perceived as not existing. *

This ambiguity about the meaning of existence applies only to files with sequential organization. For files with partitioned organization, only the file as a whole is created by control language; the individual members are created by program action. This means that existence has its natural meaning for PDS members, and that it is possible to create a PDS member containing no characters.

CMS does not allow the existence of files containing no characters, and it is not possible to create such a file.

Miscellaneous differences from UNIX operating systems

The following section lists some additional features of UNIX operating systems and UNIX I/O that some programmers expect to be available on the 370 systems. These features are generally foreign to the 370 environment and impractical to implement. Code that expects the availability of these features is not portable to the 370 no matter how successfully it runs on other architectures.

- □ UNIX operating systems and many other systems support single-character unbuffered terminal I/O in which characters can be read from a terminal one at a time and may not appear on the screen until echoed by the program. This sort of full-duplex protocol is not supported by 370 terminal controllers or operating systems.
- □ Many programs assume that screen formatting is controlled by standard control sequences, such as those used by the DEC VT100 and similar terminals. The common 370 terminal architecture (the 3270 family) bears no similarities whatsoever to that of terminals commonly used with UNIX operating systems. Although MVS and CMS support the use of terminals similar to the VT100, they are not commonly used and are not supported well enough to make running UNIX full-screen applications on them a viable proposition.
- □ The 370 operating systems offer little or no support for the use of files by more than one program simultaneously. Programs that want to do file sharing must issue system calls to synchronize with each other and obey a number of restrictions in the way the shared files are used. Because common system programs such as compilers, linkers, and copy utilities do not attempt to synchronize in this way, attempting to share files with these programs is unsafe.

* This is the interpretation used in the SAS/C implementation.

- There is no MVS or CMS concept corresponding to the pipe. Data are usually passed from program to program by means of temporary files.
- In general, the size of a file cannot be determined in any way other than by reading the entire file. The MVS and CMS equivalents of directories and inodes record the file size in terms of either the number of records or the hardware address of the end of file.
- In UNIX operating systems and many other systems, the time at which a file was last written or accessed can be determined easily. Under MVS, this information is not recorded. For PDS members, popular editors frequently store such information in a control area of the file, but this information is both difficult to access and not reliable, because updates by programs that do not support this feature (such as linkers and copy utilities) do not maintain the data appropriately.

Summary of 370 I/O characteristics

The following list describes the characteristics of 370 files (without any special reference to the C language). The points are listed in the same order as the corresponding points for the UNIX and ISO/ANSI I/O models as presented earlier:

- Many different kinds of files are possible. In general, files are not simply sequences of characters, as an additional structure is imposed by grouping the characters of a file into records. Whether a file can contain no characters depends on the file type.
- The records of a file are separated by logical or physical gaps. Control characters have no special significance and never serve as record or line separators.
- It is not possible to position a file to a particular character. Usually, it is possible to position efficiently to a particular record, but records are frequently identified by hardware-oriented addresses rather than by record numbers.
- Most files have restrictions on record length and file size, depending on their attributes. It is frequently necessary to write padding characters to force a file to conform to these attributes.
- Files can be opened for reading or writing or both. Usually, it is not possible to open a file so that new characters can be added and old characters replaced. It depends on file type and how it is accessed whether replacing an existing character truncates the file or leaves its length unchanged.

SAS/C I/O Concepts

In an ideal C implementation, C I/O would possess all three of the following properties:

- It would be compatible with UNIX operating systems.
- It would be efficient.
- It would work with all kinds of files.

For the reasons detailed in “IBM 370 I/O Concepts” on page 3-7, C I/O on the 370 cannot support all three of these properties simultaneously. The library provides several different kinds of I/O to allow the programmer to select the properties that are most important.

The library offers two separate I/O packages:

- Standard I/O is defined according to the ISO/ANSI standard. It is efficient and works with all kinds of files, but in many ways it is not compatible with UNIX operating systems. For files with suitable attributes (as described in the next section, “Standard I/O”), standard I/O is efficient and compatible with UNIX operating systems, but many files are not of this sort, especially files under MVS. Besides the ISO/ANSI standard I/O functions, the library provides a number of augmented functions, which provide non-portable access to mainframe-specific functionality.
- UNIX style I/O is compatible with UNIX low-level I/O and supports all types of files, but it is generally not efficient.

Details on both of these I/O packages are presented in the following sections. *

Standard I/O

Standard I/O is implemented by the library in accordance with its definition in the C Standard. A file may be accessed as a binary stream, in which case all characters of the file are presented to the program unchanged. When file access is via a binary stream, all information about the record structure of the file is lost. On the other hand, a file may be accessed as a text stream, in which case record breaks are presented to the program as new-line characters ('\n'). When data are written to a text file and then read, the data may not be identical to what was written because of the need to translate control characters and possibly to pad or split text lines to conform to the attributes of the file.

Besides the I/O functions defined by the Standard, several augmented functions are provided to exploit 370-specific features. For instance, the `afopen` function is provided to allow the program to specify 370-dependent file attributes, and the `afread` routine is provided to allow the program to process records that may include control characters. Both standard I/O functions and augmented functions may be used with the same file.

Library access methods

The low-level C library routines that interface with the MVS or CMS physical I/O routines are called *C library access methods* or *access methods* for short. (The term *MVS access method* always refers to access methods such as BSAM, BPAM, and VSAM to avoid confusion.) Standard I/O supports five library access methods: `''term''`, `''seq''`, `''rel''`, `''kvs''`, and `''fd''`.

When a file is opened, the library ordinarily selects the access method to be used. However, when you use the `afopen` function to open a file, you can specify one of these particular access methods.

- The library uses the `''term''` access method to perform terminal I/O; this access method applies only to terminal files. (See “Terminal I/O” on page 3-51 for more information on this access method.)
- The `''rel''` access method is used for nonterminal files whose attributes permit them to support UNIX file behavior when accessed as binary streams.
- The `''kvs''` access method is used for VSAM files when access is via the SAS/C nonstandard keyed I/O functions. (See “Using VSAM Files” on page 3-72.)
- The `''fd''` access method is used for files in the OpenEdition hierarchical file system.
- The `''seq''` access method is used with all text streams and for binary streams that cannot support the `''rel''` access method, except when `''fd''` is used.

The `''rel''` access method Under MVS, the `''rel''` access method can be used for files with sequential organization and RECFM F, FS, or FBS. (The limitation to sequential organization means that the `''rel''` access method cannot be used to process a PDS member.) Under CMS, the `''rel''` access method can be used for disk files with RECFM F. The `''rel''` access method is designed to behave like UNIX disk I/O:

* Two other I/O packages are provided: CMS low-level I/O, defined for low-level access to CMS disk files, and OS low-level I/O, which performs OS-style sequential I/O. These forms of I/O are nonportable and are discussed in Chapter 2, “CMS Low-Level I/O Functions,” and Chapter 3, “MVS Low-Level I/O Functions,” in *SAS/C Library Reference, Volume 2.*

- All characters are addressable by their character number. It is possible to position efficiently to any character.
- It is possible to replace characters before the end of file and add new data after the end of file without closing and reopening the file. A file never becomes smaller, except when the open call requests that the file's previous contents be discarded.

Because of the nature of the 370 file system, complete UNIX compatibility is not possible. In particular, the following differences still apply:

- It is not possible to create a file containing no characters using the `''rel''` access method.
- Padding null characters '(`\0`)' will be added at the end of file, if necessary to complete a record when the file is closed. If you define a file processed by the `''rel''` access method to have a record length of 1, you can avoid this padding.

The `''kvs''` access method The `''kvs''` access method processes any file opened with the extension open mode `''k''` (indicating keyed I/O). This access method is discussed in more detail in "Using VSAM Files" on page 3-72.

The `''fd''` access method The `''fd''` access method processes any file residing in the OpenEdition MVS hierarchical file system. These files are fully compatible with UNIX. In files processed with the `''fd''` access method, there is no difference between text and binary access.

The `''seq''` access method The `''seq''` access method processes a nonterminal non OpenEdition file if any one of the following apply:

- the file is to be accessed as text
- the file is not suitable for `''rel''` access
- the use of the `''seq''` access method is specifically requested.

In general, the `''seq''` access method is implemented to use efficient native interfaces, forsaking compatibility with UNIX operating systems where necessary. Some specific incompatibilities are listed here:

- The operating system being used and the file type determine whether an empty file can be created.
- File positions are represented in a way natural to the file type and the operating system, not as character numbers. The ISO/ANSI `fsetpos` and `fgetpos` functions are fully supported, except for certain files with unusual attributes such as multivolume disk files. (See Tables 3.5 and 3.6 for a complete list of restricted file types.)

  The `fseek` and `ftell` functions are supported only for text streams. This restriction is necessary because the C Standard requires that the file position be defined as a relative character number for binary streams, which cannot be efficiently determined on 370 systems. If an application requires access to binary data by character number, it should be either restricted to using files that can be processed by the `''rel''` access method, or it should use the UNIX style I/O package.
- Padding of lines for a text stream and padding at end of file for a binary stream frequently occurs. The `afopen` function gives you some control over the way padding is performed.
- For some files, changing data within a file causes the file to be truncated at the point of change; that is, all data following the change is lost. This behavior is system and file type dependent. With `afopen`, the program can inform the library of any dependence on truncation or lack of truncation. For CMS disk files, truncation is optional and you can use `afopen` to indicate whether truncation should occur.

UNIX style I/O

The library provides UNIX style I/O to meet two separate needs:

- □ to support the same functions as UNIX low-level I/O: `open`, `read`, `write`, `lseek`, and `close`. This allows programs that use these functions to run easily with the SAS/C library.
- □ to support seeking by character number for all files* whether or not this is convenient and efficient to implement. This allows programs that require this property to execute successfully, although more slowly, with the SAS/C library.

As a result of the second property, UNIX style I/O is less efficient than standard I/O for the same file, unless the file is suitable for `''rel''` access, or it is in the OpenEdition hierarchical file system. In these cases, there is little additional overhead.

For files suitable for `''rel''` access, UNIX style I/O simply translates I/O requests into corresponding calls to standard I/O routines. Thus, for these files there is no decrease in performance.

For files in the OpenEdition hierarchical file system, UNIX style I/O calls the operating system low-level I/O routines directly. For these files, use of standard I/O by UNIX style I/O is completely avoided.

For other files, UNIX style I/O copies the file to a temporary file using the `''rel''` access method and then performs all requested I/O to this file. When the file is closed, the temporary file is copied back to the user's file, and the temporary file is then removed. This means that UNIX style I/O for files not suitable for `''rel''` access has the following characteristics:

- □ The necessity of copying the data makes UNIX style I/O somewhat inefficient. However, after the copying is done, file operations are efficient, except for `close` of an output file, when all the data must be copied back. As an optimization, input data are copied from the user's file only as necessary, rather than copying all the data when the file is opened.
- □ If there is a system failure while a file is being processed with UNIX style I/O, the file is unchanged, because no data are written to an output file until the file is closed.
- □ It is possible for the processing of a file with UNIX style I/O to fail if there is not enough disk space available to make a temporary copy.
- □ Because UNIX style I/O completely rewrites an output file when the file is closed, file truncation does not occur. That is, characters are not dropped as a result of updates before the end of file.

All of the discussion within this section assumes that the user's file is accessed as a binary file: that is, without reference to any line structure. Occasionally, there are programs that want to use this interface to access a file as a text file. (Most frequently, such programs come from non UNIX environments like the IBM PC.)

As an extension, the library supports using UNIX style I/O to process a file as text. However, file positioning by character number is not supported in this case, and no copying of data takes place. Instead, UNIX style I/O translates I/O requests to calls equivalent to standard I/O routines.

---

* The library connects the use of the UNIX low-level I/O interface and the ability to do seeking by character number because UNIX documentation has traditionally stressed that seeking by character number is not guaranteed when standard I/O is used. The *UNIX Version 7 Programmer's Manual* states that the file position used by standard I/O "is measured in bytes only on UNIX; on some other systems it is a magic cookie."

Note that UNIX style I/O represents open files as small integers called file descriptors. Unlike UNIX, with MVS and CMS, file descriptors have no inherent significance. Some UNIX programs assume that certain file descriptors (0, 1, and 2) are always associated with standard input, output, and error files. This assumption is nonportable, but the library attempts to support it where possible. Programs that use file 0 only for input, and files 1 and 2 only for output, and that do not issue seeks to these files, are likely to execute successfully. Programs that use these file numbers in other ways or that mix UNIX style and standard I/O access to these files are likely to fail.

UNIX operating systems follow specific rules when assigning file descriptors to open files. The library follows these rules for OpenEdition files and for sockets. However, MVS or CMS files accessed using UNIX I/O are assigned file descriptors outside of the normal UNIX range to avoid affecting the number of OpenEdition files or sockets the program can open. UNIX programs that use UNIX style I/O to access MVS or CMS files may therefore need to be changed if they require the UNIX algorithm for allocation of file descriptors.

370 Perspectives on SAS/C Library I/O

This section describes SAS/C I/O from a 370 systems programmer's perspective. In contrast to the other parts of this chapter, this section assumes some knowledge of 370 I/O techniques and terminology.

MVS I/O implementation

Under MVS, the five C library access methods are implemented as follows:

- The **`''term''`** access method uses TPUT ASIS to write to the terminal and TGET EDIT to read from the terminal. Access to SYSTERM in batch is performed using QSAM.
- The **`''seq''`** access method uses BSAM and BPAM for both input and output. VSAM is used for access to VSAM ESDS and KSDS data sets.
- The **`''rel''`** access method uses XDAP and BSAM. XDAP is used for input and to update all blocks of the file except the last block. BSAM is used to update the last block of the file or to add new blocks. VSAM is used to access VSAM relative record data sets, and DIV is used to access VSAM linear data sets.
- The **`''kvs''`** access method uses VSAM for all operations.
- The **`''fd''`** access method uses OpenEdition service routines for all operations.

Although BDAM is not used by the **`''rel''`** access method, direct organization files that are normally processed by BDAM are supported, provided they have fixed-length records and no physical keys.

CMS I/O implementation

The C library access methods are implemented under CMS as follows:

- The **`''term''`** access method uses TYPLIN or LINEWRT to write to the terminal and WAITRD or LINERD to read from the terminal.
- The **`''seq''`** access method uses device-dependent techniques. For CMS disk files, it uses FSCB macros (FSREAD, FSWRITE, and so on). For access to shared files, it uses the CSL DMSOPEN, DMSREAD, and DMSWRITE services. For access to shared file system directories, it uses the DMSOPDIR and DMSGETDI services.

  For spool files, it uses CMS native macros such as RDCARD. For tape files, filemode 4 disk files, and files on OS disks, it uses simulated MVS BSAM. VSAM KSDS and ESDS data sets are processed using simulated VSE/VSAM.

- The **''rel''** access method uses FSCB macros. Where appropriate, it creates sparse CMS files. VSAM RRDS data sets are processed using simulated VSE VSAM.
- The **''kvs''** access method uses VSE VSAM for all operations.

File attributes for ''rel'' access under MVS

Under MVS, a file can be processed by the **''rel''** access method if it is not a PDS or PDS member, and if it has RECFM F, FS, or FBS. These record formats ensure that there are no short blocks or unfilled tracks in the file, except the last, and make it possible to reliably convert a character number into a block address (in CCHHR form) for the use of XDAP. Use of **''rel''** may also be specified for regular files in the OpenEdition file system (in which case the **''fd''** access method is used).

If the LRECL of an FBS file is 1, then an accurate end-of-file pointer can be maintained without adding any padding characters. Because of the use of BSAM and XDAP to process the file, use of this tiny record size does not affect program efficiency (data are still transferred a block at a time). However, it may lead to inefficient processing of the file by other programs or languages, notably ones that use QSAM.

File attributes for "rel" access under CMS

Under CMS, a file can be processed by the **''rel''** access method if it is a CMS disk file (not filemode 4) with RECFM F. Use of RECFM F ensures that a character number can be converted reliably to a record number and an offset within the record.

If the LRECL of a RECFM F file is 1, then an accurate end-of-file pointer can be maintained without ever adding any padding characters. Because the file is processed in large blocks (using the multiple record feature of the FSREAD and FSWRITE macros), use of this tiny record size does not affect program efficiency. Nor does it lead to inefficient use of disk space, because the files are physically blocked according to the minidisk block size. However, it may lead to inefficient processing of the file by other programs or languages that process one record at a time.

Temporary files under MVS

Temporary files are created by the library under two circumstances.

- They are created if the program calls the **tmpfile** function.
- They are created if the program uses UNIX style I/O and it is necessary to copy a file.

A program can create more than one temporary file during its execution. Each temporary file is assigned a temporary file number, starting sequentially at 1.

One of two methods is used to create the temporary file whose number is *nn.* First, a check is made for a SYSTMP*nn* DD statement. If this DDname is allocated and defines a temporary data set, then this data set is associated with the temporary file. If no SYSTMP*nn* DDname is allocated, the library uses dynamic allocation to create a new temporary file whose data set name depends on the file number. (The system is allowed to select the DDname, so there is no dependency on the SYSTMP*nn* style of name.) The data set name depends also on information associated with the running C programs, so that several C programs can run in the same address space without conflicts occurring between temporary filenames.

If a program is compiled with the **posix** compiler option, then temporary files are created in the OpenEdition hierarchical file system, rather than as MVS temporary files. The OpenEdition temporary files are created in the **/tmp** HFS directory.

Temporary files are normally allocated using a unit name of VIO and a space allocation of 50 tracks. The unit name and default space allocation can be changed by a site, as described in the SAS/C installation instructions. If a particular application requires a larger space allocation than the default, use of a SYSTMP*nn* DD statement specifying the required amount of space is recommended.

Temporary files under CMS

Temporary files are created by the library under two circumstances.

- □ They are created if the program calls the `tmpfile` function.
- □ They are created if the program uses UNIX style I/O and it is necessary to copy the file.

A program can create more than one temporary file during its execution. Each temporary file is assigned a temporary file number, starting sequentially at 1.

One of two methods is used to create the temporary file whose number is *nn*. First, a check is made for a FILEDEF of the DDname SYSTMP *nn*. If this DDname is defined, then it is associated with the temporary file. If no SYSTMP*nn* DDname is defined, the library creates a file whose name has the form $$$$$$*nn* $$$$*xxxx*, where *nn* is the temporary file number, and the *xxxx* part of the filetype is associated with the calling C program. This naming convention allows several C programs to execute simultaneously without conflicts occurring between temporary filenames.

Temporary files are normally created by the library on the write-accessed minidisk with the most available space. Using FILEDEF to define a SYSTMP*nn* DDname with another filemode allows you to use some other technique if necessary.

Be aware that these temporary files are not known to CMS as temporary files. Therefore, they are not erased if a program terminates abnormally or if the system fails during its execution.

VSAM usage and restrictions

The SAS/C library supports two different kinds of access to VSAM files: standard access, and keyed access. Standard access is used when a VSAM file is opened in text or binary mode, and it is limited to standard C functionality. Keyed access is used when a VSAM file is used in keyed mode. Keyed mode is discussed in detail in "Using VSAM Files" on page 3-72.

Any kind of VSAM file may be used via standard access. Restrictions apply to particular file types, for example, a KSDS may not be opened for output using standard I/O.

- □ The library supports VSAM ESDS data sets as it supports other sequentially organized file types. A VSAM ESDS can be accessed as a text stream or a binary stream using standard I/O or UNIX style I/O. A VSAM ESDS is not suitable for processing by the `''rel''` access method because it is not possible, given a character position, to determine the RBA (relative byte address) of the record containing the character.
- □ The library supports VSAM KSDS data sets for input only. Output is not supported for standard access because the C I/O routines are unaware of the existence of keys and cannot guarantee that new records are added in key order. Use keyed access instead. Also, file positioning with `fseek` or `fsetpos` is not supported, because records are ordered by key, and it is not possible to transform a key value into the file position formats used for other library file types. When a KSDS is used for input, records are always presented to the program in key order, not in the order of their physical appearance in the data set. Note that KSDS output is available when keyed access is used.
- □ The library supports VSAM RRDS data sets for access via the `''rel''` access method only. Only RRDS data sets with a fixed record length are supported. As with all other files accessed via `''rel''`, file positioning using `fseek` and `ftell` are fully supported.

- The library supports VSAM linear data sets that are also known as Data-in-Virtual (DIV) objects. You must access a DIV object as a binary stream, and you must use the `''rel''` access method. As with all other files accessed via `''rel''`, file positioning using `fseek` and `ftell` are fully supported.

VSAM ESDS, KSDS and RRDS files are processed using a single RPL. Move mode is used to support spanned records. A VSAM file cannot be opened for write only (open mode `''w''`) unless it was defined by Access Method Services to be a reusable file.

Choosing I/O Techniques and File Organization

Because of the wide variety of I/O techniques and 370 file types, it is not always easy to select the right combination for a particular application. Also, the considerations differ for new applications and for existing applications ported from another environment.

New Applications

Recommendations for I/O in new programs depend on whether the program needs to run on other systems. For portable applications, the following guidelines are recommended:

- If OpenEdition is available on your system, use OpenEdition files wherever appropriate. Because OpenEdition files implement UNIX semantics, I/O to these files is more portable than I/O to traditional MVS files.
- Use standard I/O rather than UNIX style I/O. Because standard I/O is endorsed by the ISO/ANSI standard, it is more portable than UNIX style I/O. It is also more efficient than UNIX style I/O on 370 systems, and it is not appreciably slower on most other systems.
- Open a file for text access if the file will be processed as a series of lines. Open it for binary access if it will be processed as a series of characters.
- If file positioning is required for text applications, use the `fseek` and `ftell` functions, which are more widely available than `fsetpos` or `fgetpos`. Note that you cannot use file positions arithmetically with these functions. (You may be forced to use `fsetpos` and `fgetpos` rather than `fseek` and `ftell` if you need to support very large files.)
- If file positioning is required for binary applications, use UNIX style I/O, use `fsetpos` and `fgetpos`, or restrict the application to using files suitable for `''rel''` access. The advantage of UNIX style I/O is that it is applicable to most files and is somewhat portable. The advantage of `fsetpos` and `fgetpos` is that they are defined by the C Standard, so they are portable. The advantage of restricting the application to files suitable for `''rel''` access is that maximum efficiency is achieved with the most portable interface. If the file is only used by C programs (for example, if the file is a work file, or if it is not accessed by the outside world), then requiring suitable file attributes is clearly the best solution.

If your application does not have to be portable, there are several additional possibilities. Note, however, that even if portability is not a requirement, one of the portable techniques described earlier may still be most appropriate.

- If your application needs to process data one record at a time, consider using the augmented standard I/O routines `afread` and `afwrite`. These routines are especially useful if the records may contain control characters (which makes standard I/O text access inappropriate).
- For nonportable applications, use `fsetpos` and `fgetpos` rather than `fseek` and `ftell` for file positioning. These routines have fewer restrictions and their results are more easily interpreted.

- □ If efficiency is a major consideration, you may want to use low-level I/O, as described in Chapter 2, "CMS Low-Level I/O Functions," and Chapter 3, "MVS Low-Level I/O Functions," in *SAS/C Library Reference, Volume 2.*
- □ Avoid UNIX style I/O.

Existing Applications

For existing applications, the choices are more difficult. With an existing program, you may be forced to choose between rewriting the program's I/O routines, accepting poor performance, and restricting use of the program to certain types of files. The following is a partial list of the considerations:

- □ If the program uses standard I/O and processes a file as text, changes are required if the file position must be a relative character number. Changes may also be required if the program reads or writes control characters, or is sensitive to the presence or absence of trailing blanks.
- □ If the program uses standard I/O, processes a file as binary, and uses `fseek` and `ftell` for file positioning, you must modify the program or restrict it to use only the `''rel''` access method. (Such an application could be modified to use UNIX style I/O or to use `fsetpos` and `fgetpos` for positioning.) Further modifications or restrictions on file type may be required if the program cannot tolerate the addition of trailing nulls at end of file.
- □ If a program uses standard I/O to modify a file and requires that later data in the file be preserved, the program must be restricted to certain file types (for example, VSAM or CMS-format disk files) or be modified to use UNIX style I/O.
- □ A program that uses standard I/O to append data to an existing file cannot update an MVS PDS member. Such a program must be restricted to use of files with sequential organization or (provided binary access is used) converted to use UNIX style I/O.
- □ If the program uses UNIX style I/O and processes a file as binary, it usually executes without modification. Performance is improved if the file is suitable for `''rel''` access, because then the file can be processed without copying. If the program does not depend on some of the details of UNIX style I/O (for instance, if it is not sensitive to the exact nature of file positions), it can be converted to use standard I/O for better performance.
- □ If the program uses UNIX style I/O, processes a file as text, and uses `lseek` to do file positioning, it requires substantial modification. The library does not support file positioning by character number when UNIX style I/O is used to access a file as text.
- □ If a program using either I/O package treats a file sometimes as text and sometimes as binary (that is, it interprets a new-line character as both a line separator and a physical character), the program must be modified.

Technical Summaries

This section provides detailed discussions of many of the components of C I/O, such as opening files, file positioning, and using the standard input, output, and error files. There are also sections that address I/O under OpenEdition, advanced MVS and CMS I/O facilities, and how to perform VSAM keyed I/O in C. The last section attempts to answer some of the most commonly asked I/O questions.

Standard I/O Overview

The standard I/O package provides a wide variety of functions to perform input, output, and associated tasks. It includes both standard functions and augmented functions to support 370-oriented features.

In general, a program that uses standard I/O accesses a file in the following steps:

1. Open the file using the standard function **`fopen`** or the augmented function **`afopen`**. This establishes a connection between the program and the external file. The name of the file to open is passed as an argument to **`fopen`** or **`afopen`**. The **`fopen`** and **`afopen`** functions return a pointer to an object of type **`FILE`**. (This type is defined in the header file **`<stdio.h>`**, which should be included with a **`#include`** statement by any program that uses standard I/O.) The data addressed by this **`FILE`** pointer are used to control all further program access to the file.
2. Transfer data to and from the file using any of the functions listed in this section. The **`FILE`** pointer returned by **`fopen`** is passed to the other functions to identify the file to be processed.
3. Close the file. After the file is closed, all changes have been written to the file and the **`FILE`** pointer is no longer valid. When a program terminates (except as the result of an ABEND), all files that have not been closed by the program are closed automatically by the library.

For convenience, three standard files are opened before program execution, accessible with the **`FILE`** pointers **`stdin`**, **`stdout`**, and **`stderr`**. These identify the standard input stream, standard output stream, and standard error stream, respectively. For TSO or CMS programs, these **`FILE`** objects normally identify the terminal, but they can be redirected to other files by use of command-line options. For programs running under the OpenEdition shell, these **`FILE`** objects reference the standard files for the program that invoked them. More information on the standard streams is available later in this section.

Standard I/O functions may be grouped into several categories. The functions in each category and their purposes are listed in Table 3.1.

Table 3.1
Standard I/O Functions

Function	Purpose
Control Functions	**control basic access to files**
`fopen`+	opens a file
`afopen`*+	opens a file with system-dependent options
`freopen`+	reopens a file
`afreopen`*+	reopens a file with system-dependent options
`tmpfile`	creates and opens a temporary file
`tmpnam`	generates a unique filename
`fflush`	writes any buffered output data
`afflush`+	forces any buffered output data to be written immediately
`fclose`+	closes a file
`setbuf`+	changes stream buffering
`setvbuf`+	changes stream buffering

continued

Table 3.1 (*continued*)

Function	**Purpose**
Character I/O Functions	**read or write single characters**
`fgetc`	reads a character
`getc`	reads a character (macro version)
`ungetc`	pushes back a previously read character
`getchar`	reads a character from `stdin`
`fputc`	writes a character
`putc`	writes a character (macro version)
`putchar`	writes a character to `stdout`
String I/O Functions	**read or write character strings**
`fgets`	reads a line into a string
`gets`	reads a line from `stdin` into a string
`fputs`	writes a string
`puts`	writes a line to `stdout`
Array I/O Functions	**read or write arrays or objects of any data type**
`fread`	reads one or more data elements
`fwrite`	writes one or more data elements
Record I/O Functions	**read or write entire records**
`afread`*	reads a record
`afread0`*	reads a record (possibly length 0)
`afreadh`*	reads the initial part of a record
`afwrite`*	writes a record
`afwrite0`*	writes a record (possibly length 0)
`afwriteh`*	writes the initial part of a record
Formatted I/O Functions	**easily read or write formatted data**
`fprintf`	writes one or more formatted items
`printf`	writes one or more formatted items to `stdout`
`sprintf`	formats items into a string
`snprintf`*	formats items into a string (with maximum length)
`fscanf`	reads one or more formatted items

continued

Table 3.1 (*continued*)

Function	**Purpose**
`scanf`	reads one or more formatted items from `stdin`
`sscanf`	obtains formatted data from a string
`vfprintf`	writes formatted data to a file
`vprintf`	writes formatted data to standard output stream
`vsprintf`	writes formatted data to a string
`vsnprintf`	writes formatted data to a string (with maximum length)
File Positioning Functions	**interrogate and change the file position**
`fseek`	positions a file
`fsetpos`	positions a file
`rewind`	positions a file to the first byte
`ftell`	returns current file position for `fseek`
`fgetpos`	returns current file position for `fsetpos`
Keyed Access Functions	**read, write and position a keyed stream**
`kdelete`*+	delete a record from a keyed file
`kgetpos`*+	return position of current keyed file record
`kinsert`*+	add a new record to a keyed file
`kreplace`*+	replace a new record in a keyed file
`kretrv`*+	retrieve a record from a keyed file
`ksearch`*+	search for a record in a keyed file
`kseek`*+	reposition a keyed file
`ktell`*+	return RBA of current record of keyed file
Error-Handling Functions	**test for and continue execution after I/O errors and other I/O conditions**
`feof`+	tests for end of file
`ferror`+	tests for error
`clearerr`+	resets previous error condition
`clrerr`*+	resets previous error condition
File Inquiry Functions	**obtain information about an open file at execution time**
`fattr`*+	returns file attributes

continued

Table 3.1 (*continued*)

Function	Purpose
`fileno`*	returns file number
`ffixed`*+	tests whether a file has fixed length records
`fnm`*+	returns the name of a file
`fterm`*+	tests whether a file is the user's terminal

*These functions are not defined in the ANSI standard. Programs that use them should include **`lcio.h`** rather than **`stdio.h`**.

+These functions may be used with files opened for keyed access.

UNIX Style I/O Overview

The UNIX style I/O package is designed to be compatible with UNIX low-level I/O, as described in previous sections. When you use UNIX style I/O, your program still performs the same three steps (open, access, and close) as those performed for standard I/O, but there are some important distinctions.

- To open a file using UNIX style I/O, you call **`open`** or **`aopen`**. (**`aopen`** is not compatible with UNIX operating systems but permits the program to specify 370-dependent file processing options.) The name of the file to open is passed as an argument to **`open`** or **`aopen`**.
- **`open`** and **`aopen`** return an integer called the file number (sometimes file descriptor). The file number is passed to the other UNIX style functions to identify the file. It indexes a table containing information used to access all files accessed with UNIX style I/O. Be sure to use the right kind of object to identify a file: a **`FILE`** pointer with standard I/O, but an integer file number with UNIX style I/O.

By convention, UNIX assigns the file numbers 0, 1, and 2 to the standard input, output, and error streams. Some programs use UNIX style I/O with these file numbers in place of standard I/O to **`stdin`**, **`stdout`**, and **`stderr`**, but this practice is nonportable. The library attempts to honor this kind of usage in simple cases, but for the best results the use of standard I/O is recommended.

UNIX style I/O offers fewer functions than standard I/O. No formatted I/O functions or error-handling functions are provided. In general, programs that require elaborately formatted output or control of error processing should, where possible, use standard I/O rather than UNIX style I/O. Some UNIX style I/O functions, such as **`fcntl`** and **`ftruncate`** are supported only for files in the OpenEdition hierarchical file system.

The functions supported by UNIX style I/O and their purposes are listed in Table 3.2. Note that the **`aopen`** function is not defined by UNIX operating systems. Also note that some POSIX-defined functions, such as **`ftruncate`**, are not implemented by all versions of UNIX.

Table 3.2
UNIX Style I/O Functions

Function	Purpose
Control Functions	
`aopen`*	opens a file with system-dependent options
`close`+	closes a file
`creat`	creates a file and opens it for output
`dup`-	duplicates a file descriptor
`dup2`-	duplicates a file descriptor
`fcntl`+-	controls file options and attributes
`fdopen`+-	associates a file descriptor with a FILE pointer
`fsync`	forces output to be written to disk
`ftruncate`-	truncates a file
`mkfifo`-	creates an OpenEdition FIFO special file
`mknod`-	creates an OpenEdition special file
`pipe`-	creates an OpenEdition pipe
Character I/O Functions	
`read`+	reads characters from a file
`write`+	writes characters to a file
File Positioning Functions	
`lseek`	positions a file
File Inquiry Functions	
`ctermid`	returns the terminal filename
`isatty`+	tests whether a file is the user's terminal
`ttyname`	returns the name of the terminal associated with a file descriptor

*This function is not defined by UNIX operating systems.
+This function is also supported for sockets.
-This function is supported only for OpenEdition files.

Opening Files

Although there are several different functions that you can call to open a file (for example, `fopen`, `afopen`, and `open`), they all have certain points of similarity. The filename (or pathname) and an open mode are passed as arguments to each of these functions. The filename identifies the file to be opened. The open mode defines how the file will be processed. For example, the function call `fopen(''input'', ''rb'')` opens the file whose name is `input` for binary read-only processing.

Some of the open functions enable the caller to specify a C library access method name, such as `''rel''`, and access method parameters (or amparms) such as `''recfm=v''`. Access method parameters allow the program to specify system or access-method-dependent information such as file characteristics (for example, record format) and processing options (for example, the number of lines per page). The details for each of these specifications are described in this section.

General filename specification

The general form of a SAS/C filename is [//] ***style*:*name***, where the portion of the name before the colon defines the filename style, and the portion after the colon is the name proper. For example, the style **ddn** indicates that the filename is a DDname, while the style **cms** indicates that the filename is a CMS fileid or device name.

Note: The **//** before the rest of the pathname is optional, except for programs compiled with the **posix** option. See "OpenEdition I/O Considerations" on page 3-56 for a discussion of filenames for **posix**-compiled programs.

The ***style*:** part of the filename is optional. If no style is specified, the style is chosen as follows:

- □ If the pathname begins with a **//** prefix, the default style is **tso** in MVS, or **cms** in CMS.
- □ If you define the external variable **_style** with an initial value, then that value is used as the style. (See Chapter 9 in the *SAS/C Compiler and Library User's Guide* for more information on the external variable **_style**.) For instance, if the initial value of **_style** is **tso**, then the filename XYZ.DATA is interpreted as **tso:xyz.data**.
- □ If no initial value for **_style** is defined, the default style is **ddn** under MVS, and **cms** under CMS. This means that by default, filenames are interpreted as DDnames under MVS and as fileids or device names under CMS.

As an aid to migration of programs between MVS and CMS, filenames oriented toward one system, such as **cms:user maclib** and **tso:output.list**, are accepted by the other system when a clearly equivalent filename can be established. (See the next section for details.)

The rules just described apply only to programs that are not compiled with the **posix** compiler option. For **posix**-compiled programs, all pathnames are treated as hierarchical file system names, unless they are preceded by the **//** prefix, even if they appear to contain a style prefix.

Filename specification under MVS

The library supports four primary styles of filenames under MVS: **ddn**, **dsn**, **tso**, and **hfs**. A **ddn**-style filename is a DDname, a **dsn**-style filename is a data set name in JCL syntax, a **tso**-style filename is a data set name in TSO syntax, and an **hfs**-style filename is a pathname referencing the OpenEdition hierarchical file system.

ddn-style filenames A filename in **ddn** style is a valid DDname, possibly preceded by leading white space. The filename can be in uppercase or lowercase letters, although it is translated to uppercase letters during processing. The following alternate forms are also allowed, permitting access to PDS members and to the TSO terminal:

```
*
*ddname
ddname*
ddname(member-name)
```

A **ddn**-style filename of ***** always references the user's TSO terminal. If you use this filename in a batch job, it references the SYSTERM DDname, if the file is being opened for output or append. (See "Open modes" on page 3-33 for more information.) The filename ***** is not supported for input in batch. For a program executing under the

OpenEdition shell, a **ddn**-style filename of ***** is interpreted as referencing **/dev/tty** in the hierarchical file system.

A **ddn** style filename of ***ddname** references the terminal for a TSO session or the DDname indicated for a batch job. For example, the filename ***sysin** requests use of the terminal under TSO or of the DDname SYSIN in batch.

A **ddn**-style filename of **ddname*** references the indicated DDname, if that DDname is defined. If the DDname is not defined, **ddname*** references the TSO terminal or SYSTERM in batch (for an open for output or append). For example, the filename **LOG*** requests the use of the DDname LOG, if defined, and otherwise, the user's TSO terminal.

A **ddn**-style filename of **ddname(*member-name*)** references a member of the PDS identified by the DDname. For example, the filename **SYSLIB(fcntl)** requests the member FCNTL of the data set whose DDname is SYSLIB. If the DD statement also specifies a member name, the member name specified by the program overrides it.

With the availability of OpenEdition, another **ddn**-style filename is possible:

```
ddname/filename
```

Here, **ddname** is a valid MVS filename, and **filename** is a valid POSIX filename (not containing any slashes).

For more information on this form, see "Using HFS directories and PDS members interchangeably" on page 3-57.

Note: Programs invoked via the OpenEdition **exec** system call do not ordinarily have access to DD statements. A SAS/C extension allows environment variables to be used in place of DD statements, as described in "OpenEdition I/O Considerations" on page 3-56.

dsn-stylefilenames A filename in **dsn** style is a valid, fully qualified data set name (possibly including a member name), optionally preceded by leading white space. The data set name can be in uppercase or lowercase letters, although it is translated to uppercase letters during processing. The data set name must be completely specified; that is, there is no attempt to prefix the name with a userid, even for programs running under TSO. (Programs that want to have the prefix added should use the **tso** filename style.) For more information on data set names and their syntax, consult the IBM manual *MVS/ESA JCL Reference.*

The following alternate forms for **dsn**-style names are also allowed, permitting access to temporary data sets, the TSO terminal, DUMMY files, and SYSOUT files:

```
*
nullfile
sysout=classtmpname
```

A **dsn**-style filename of ***** always references the user's TSO terminal. If this filename is used in a batch job, it references the SYSTERM DDname, if the file is being opened for output or append. (See "Open modes" on page 3-33.) The filename ***** is not supported for input in batch. For a program running under the OpenEdition shell, ***** is interpreted as referencing **/dev/tty**.

A **dsn**-style filename of **nullfile** references a DUMMY (null) data set. Reading a DUMMY data set produces an immediate end of file; data written to a DUMMY data set are discarded.

A **dsn**-style filename of **sysout=*class*** references a SYSOUT (printer or punch) data set of the class specified. The ***class*** must be a single alphabetic or numeric character, or an asterisk.

A **dsn**-style filename of ***&tmpname*** references a new temporary data set, whose name is ***&tmpname***. The name is limited to eight characters.

tso-style filenames A filename in **tso** style is a data set name (possibly including a member name) specified according to TSO conventions, optionally preceded by leading white space. The data set name can be in uppercase or lowercase letters, although it is translated to uppercase letters during processing. If the data set name is not enclosed in single quotation marks, the name is prefixed with the user's TSO prefix (normally the userid), as defined by the TSO PROFILE command. If the data set name is enclosed in single quotation marks, the quotes are removed and the result is interpreted as a fully qualified data set name. For more information on TSO data set names and their syntax, consult the IBM manual *TSO Extensions User's Guide.*

Note: **tso**-style filenames are not guaranteed to be available, except for programs executing under TSO. If you attempt to open a **tso**-style filename in MVS batch (or under the OpenEdition shell), the userid associated with the batch job is used as the data set prefix. Determining the userid generally requires RACF or some other security product to be installed on the system. If the userid cannot be determined, the open call will fail.

The following alternate forms for **tso**-style names are also allowed, permitting access to the TSO terminal and DUMMY files.

```
*
'nullfile'
```

A **tso**-style filename of ***** always references the user's TSO terminal. For programs running under the OpenEdition shell, it is interpreted as referencing the HFS file called **/dev/tty**.

A **tso**-style filename of **'nullfile'** references a DUMMY data set. Reading a DUMMY data set produces an immediate end of file; data written to a DUMMY data set are discarded.

cms-style filenames For compatibility with CMS, the MVS version of the SAS/C library accepts **cms**-style filenames, where possible, by transforming them into equivalent **tso**-style filenames. See the next section, "Filename specification under CMS," for details on the format of **cms** style filenames.

A **cms**-style filename is transformed into a **tso**-style filename by replacing spaces between the filename components with periods, removing the MEMBER keyword, and adding a closing parenthesis after the member name, if necessary. Also, the filenames **cms: *** and **cms:** are interpreted as **tso: *** and **tso: 'nullfile'**, respectively. For instance, the following transformations from **cms**-style to **tso**-style names are performed:

```
cms: profile exec a1              tso: profile.exec.a1
cms: lc370 maclib (member stdio   tso: lc370.maclib(stdio)
cms: reader                       tso: reader
cms: *                            tso: *
cms:                              tso: 'nullfile'
```

hfs-style filenames A filename in **hfs** style is a pathname in the OpenEdition hierarchical file system. If the pathname begins with a **/**, the pathname is an absolute pathname, starting at the root directory. If the pathname does not begin with a **/**, it is interpreted relative to the current directory.

Note: You cannot open an HFS directory using **fopen** or **open**. The **opendir** function must be used for this purpose.

Filename specification under CMS

The library supports five primary styles of filename under CMS: `cms`, `xed`, `ddn`, `sf`, and `sfd`. A `cms`- or `xed`-style filename is a CMS fileid or device name. A `ddn`-style file is a DDname (FILEDEF or DLBL name). A `sf`-style filename is the name of a CMS shared file system file, and a `sfd`-style filename is a pattern defining a subset of a CMS shared file system directory.

The only difference between the `cms` and `xed` styles is that, if a program is running under XEDIT, use of the `xed` prefix allows reading of the file from XEDIT, rather than from disk.

cms- and xed-style filenames A filename in `cms` style is a CMS fileid or device name. You can specify fileids in one of two formats: a CMS standard format, or a compressed format. The compressed format contains no blanks, so it can be used in cases in which the presence of blanks is not allowed, such as in command-line redirections. The `xed` style permits a subset of the valid `cms` specifications, as described in "Advanced CMS I/O Facilities" on page 3-67. Here is the standard form for a `cms`-style filename:

```
filename  [filetype [filemode]] [(MEMber member-name)]]
```

The brackets indicate optional components. The filename may be preceded by white space and can be in uppercase or lowercase letters, although it is translated to uppercase letters during processing. Detailed rules for this style of filename are as follows:

- If no filetype is specified, the filetype `FILE` is assumed, unless the MEMBER keyword is present, in which case, the filetype `MACLIB` is assumed.
- If filemode is omitted or is specified as `*`, a search is made for an existing file on an accessed disk using the standard CMS search order. If no existing file can be located, and the open mode permits output, a filemode of A1 is assumed.
- You can specify a ***member-name*** only for files whose filetype is `MACLIB` or `TXTLIB`, opened for input. The keyword `MEMber` may be abbreviated to `MEM`. (The `xed` style does not allow a member name to be specified.)

Here is the compressed form for a `cms`-style filename:

```
filename [.filetype [. filemode]] [(member-name)]
```

This form of filename is interpreted in exactly the same way as the corresponding standard name. For example, `cms: freds maclib (mem smith)` and `cms: freds.maclib(smith)` are equivalent specifications. For more information on CMS fileids, consult the IBM CMS manuals listed in Chapter 1, "Introduction," in the *SAS/C Compiler and Library User's Guide.*

The following alternate forms for `cms`-style names are also allowed, permitting access to unit record devices and members of GLOBAL MACLIBs or TXTLIBs. (See "Advanced CMS I/O Facilities" on page 3-67 for a description of access to GLOBAL MACLIBs and TXTLIBs.) After each form, valid abbreviations are given. (None of these forms can be used with the `xed` style.)

Alternate Forms	**Abbreviations**
TERMINAL	TERM, *
READER	RDR
PRINTER	PRT
PUNCH	PUN, PCH
%MACLIB(member *member-name*)	%MACLIB(*member-name*)
%TXTLIB(member *member-name*)	%TXTLIB(*member-name*)

Also, an empty filename (`''''`) may be used to open a dummy file.
Note: To open a CMS disk file whose filename is the same as one of the above device names, you must specify both the filename and the filetype.

ddn-style filenames A filename in **`ddn`** style is a valid DDname, possibly preceded by white space. The filename can be in uppercase or lowercase letters, although it is translated to uppercase letters during processing. The DDname must be previously defined using the FILEDEF command (or the DLBL command for a VSAM file). The following alternate forms are also allowed, permitting access to members of MACLIBs, TXTLIBs, and MVS PDSs, and to the CMS terminal. (All forms have approximately the same meaning as under MVS.) For more information, see "Filename specification under MVS" on page 3-28.

```
*
*ddname
ddname*
ddname(member-name)
```

A **`ddn`**-style filename of **`*`** always references the user's CMS terminal.
A **`ddn`**-style filename of **`*ddname`** also references the terminal. (The DDname is never used because the terminal is always defined under CMS.)
A **`ddn`**-style filename of **`ddname*`** references the indicated DDname, if that DDname is defined. If the DDname is not defined, it references the CMS terminal. For example, the filename **`LOG*`** requests the use of the DDname LOG, if defined, and otherwise, the user's terminal.
A **`ddn`**-style filename of **`ddname(`*`member-name`*`)`** references a member of the MACLIB, TXTLIB, or MVS PDS identified by the DDname. For example, the filename **`SYSLIB(fcntl)`** requests the member FCNTL of the file whose DDname is SYSLIB. If the FILEDEF command also specifies a member name, the member name specified by the program overrides it.

tso-style filenames For compatibility with MVS, the CMS version of the library accepts **`tso`**-style filenames where possible, by transforming them into equivalent **`cms`**-style filenames. See "Filename specification under MVS" on page 3-28 for details on the format of such filenames.
A **`tso`**-style filename is transformed into a **`cms`**-style filename by removing single quotation marks, if present, and treating the resulting name as a compressed format fileid. (The result must be a valid CMS fileid or the open fails.) In addition, the

specification `tso: *` is interpreted as `cms: terminal`. For instance, the following transformations from `tso`-style to `cms`-style names are performed:

```
tso: input.data                   cms: input data
tso: parser.source.c              cms: parser source c
tso: 'sys1.maclib(dcb)'           cms: sys1 maclib (member dcb
tso: *                            cms: terminal
```

sf-style filenames A `sf`-style filename references a file in the CMS shared file system. See "Using the CMS Shared File System" on page 3-69 for detailed information on the syntax of `sf`-style filenames.

sfd-style filenames A `sfd`-style filename references a CMS shared file system directory or directory subset. See "Using the CMS Shared File System" on page 3-69 for detailed information of the syntax of `sfd`-style filenames.

Open modes

The second argument to each open routine is an *open mode*, which defines how the file will be processed. This argument is specified differently, depending on whether you are using standard I/O or UNIX style I/O, but the basic capabilities are the same.

Standard I/O open modes When you open a file using standard I/O, the open mode is a character string consisting of one to three enquoted characters. The syntax for this string is as follows:

```
r|w|a[+][b|k]
```

The first character must be `'r'`, `'w'`, or `'a'`. After the first character, a `'+'` may appear, and after the `'+'` (or after the first character, if `'+'` is omitted), `'b'` or `'k'` may appear. No blanks may appear in the string, and all characters must be lowercase letters.

The `'r|w|a'` character specifies whether the file is to be opened for reading, writing, or appending. If a `'+'` appears, both reading and writing are permitted.

If a `'b'` appears, the file is accessed as a binary stream. If a `'k'` appears, the file is accessed as a keyed stream. If neither `'b'` nor `'k'` appears, the file is accessed as text. See "Text access and binary access" on page 3-52 for detailed information on the differences between text and binary access. See "Using VSAM Files" on page 3-72 for information on keyed access.

The effect of the `'r|w|a'` specification and the `'+'` are closely linked and must be explained together.

- A file opened with open mode `'r'` or `'rb'` is a read-only file. The file must already exist. (See "File existence" on page 3-12.)
- A file opened with open mode `'rk'` is a read-only file suitable for keyed access. Records can be retrieved, but not replaced, deleted, or inserted. If the file has not been loaded, the open will fail.
- A file opened with open mode `'r+'` or `'r+b'` can be both read and written. The file must already exist. (See "File existence" on page 3-12.)
- A file opened with open mode `'r+k'` can be read, or written using keyed access, or both. All file operations are permitted. If the file has not been loaded, the open will fail.
- A file opened with open mode `'w'` or `'wb'` is a write-only file. If the file already exists, its previous contents are discarded.

- A file opened with open mode `'wk'` is a write-only file suitable for keyed access. If the file contains any records, they are erased. If the file is not defined as REUSABLE and it contains any records, the open will fail. This open mode enables records to be added to the file, but not to be retrieved, updated, or deleted.
- A file opened with open mode `'w+'` or `'w+b'` can be both read and written. If the file already exists, its contents are discarded when it is opened.
- A file opened with open mode `'w+k'` can be read, or written using keyed access, or both. If the file contains any records, they are erased. If the file is not defined as REUSABLE and it contains any records, the open will fail. All file operations are permitted.
- A file opened with open mode `'a'` or `'ab'` can only be written. If the file exists, its contents are preserved. All output is appended to the end of file.
- A file opened with open mode `'ak'` is a write-only file suitable for keyed access. Records can be inserted, but not retrieved, replaced, or deleted. (Records can be inserted at any point in the file, not just at the end.) The file does not have to be loaded in advance.
- A file opened with open mode `'a+'` or `'a+b'` can be both read and written. If the file exists, its contents are preserved. Whenever an output request is made, the file is positioned to the end of file first; however, reading may be performed at any file position. The file is initially positioned to the start of the file.
- A file opened with open mode `'a+k'` can be read and/or written using keyed access. Records can be retrieved or inserted, but not replaced or deleted. (Records can be inserted at any point in the file, not just at the end.) The file does not have to be loaded in advance.

Note: For compatibility with some PC C libraries, certain variant forms of the open mode parameter are accepted. The order of the `'+'` and the `'b'` may be reversed, and an `'a'` may appear in place of the `'b'` to request that the file be accessed as text.

UNIX style I/O open modes When you open a file using UNIX style I/O, the open mode is an integer, with open mode options indicated by the presence or absence of particular bit settings. The open mode is normally specified by ORing symbolic constants that specify the options required. For instance, the specification `O_RDONLY|O_BINARY` is used for a read-only file to be accessed as a binary stream. The symbolic constants listed here are all defined in the header file `<fcntl.h>`.

The following open mode options are supported by UNIX style I/O:

`O_RDONLY`	specifies that the file will be read but not written. If you do not specify `O_WRONLY` or `O_RDWR`, `O_RDONLY` is assumed.
`O_WRONLY`	specifies that the file will be written but not read.
`O_RDWR`	specifies that the file will be both read and written.
`O_APPEND`	specifies that the file will be positioned to the end of file before each output operation.
`O_CREAT`	specifies that if the file does not exist, it is to be created. (See "File existence" on page 3-12.) If `O_CREAT` is omitted, an attempt to open a file that does not exist fails.
`O_TRUNC`	specifies that if the file exists, the file's current contents will be discarded when the file is opened.
`O_EXCL`	is meaningful only if `O_CREAT` is also set. It excludes the use of an already existing file.
`O_NONBLOCK`	specifies the use of non-blocking I/O. This option is meaningful only for OpenEdition HFS files.

`O_NOCTTY`	specifies that the file is not to be treated as a "controlling terminal." This option is meaningful only for OpenEdition HFS files.
`O_BINARY`	specifies that the file be accessed as a binary stream. If `O_TEXT` is not specified, `O_BINARY` is assumed. (The synonym `O_RAW` is supported for compatibility with other compilers.)
`O_TEXT`	specifies that the file be accessed as a text stream.

Note: UNIX I/O does not support keyed streams.

Table 3.3 defines equivalent forms for standard I/O and UNIX style I/O open modes. Some UNIX style I/O open modes have no standard I/O equivalents.

Table 3.3 Standard I/O and UNIX Style I/O Open Modes

Standard form	UNIX style form
`'r'`	`O_RDONLY	O_TEXT`
`'rb'`	`O_RDONLY`
`'r+'`	`O_RDWR	O_TEXT`
`'r+b'`	`O_RDWR`
`'w'`	`O_WRONLY	O_CREAT	O_TRUNC	O_TEXT`
`'wb'`	`O_WRONLY	O_CREAT	O_TRUNC`
`'w+'`	`O_RDWR	O_CREAT	O_TRUNC	O_TEXT`
`'w+b'`	`O_RDWR	O_CREAT	O_TRUNC`
`'a'`	`O_WRONLY	O_APPEND	O_CREAT	O_TEXT`
`'ab'`	`O_WRONLY	O_APPEND	O_CREAT`
`'a+'`	`O_RDWR	O_APPEND	O_CREAT	O_TEXT`
`'a+b'`	`O_RDWR	O_APPEND	O_CREAT`

Library access method selection

When you use `afopen` or `afreopen` to open a file, you can specify the library access method to be used. If you use some other open routine, or specify the null string as the access method name, the library selects the most appropriate access method for you. If you specify an access method that is incompatible with the attributes of the file being opened, the open fails, and a diagnostic message is produced. Six possible access method specifications are available:

- □ A null (`''''`) access method name allows the library to select an access method.
- □ The `''term''` access method applies only to terminal files.
- □ The `''seq''` access method is primarily oriented towards sequential access. (`''seq''` may also be specified for terminal files, in which case, the `''term''` access method is automatically substituted.)
- □ The `''rel''` access method is primarily oriented toward access by relative character number. The `''rel''` access method can be used only when the open mode specifies binary access. Additionally, the external file must have appropriate attributes, as discussed in "370 Perspectives on SAS/C Library I/O" on page 3-18.
- □ The `''kvs''` access method provides keyed access to VSAM files.
- □ The `''fd''` access method provides access to OpenEdition hierarchical file system files.

When no specific access method is requested by the program, the library selects an access method as follows:

- □ **``term''** for a TSO or CMS terminal file
- □ **``kvs''** if the open mode specifies keyed access
- □ **``fd''** for a hierachical file system file
- □ **``rel''** if the open mode includes binary access and the file has suitable attributes
- □ **``seq''** otherwise.

Access method parameters

When you use **afopen**, **afreopen**, or **aopen** to open a file, you can optionally specify one or more access method parameters (or amparms). These are system-dependent options that supply information about how the file will be processed or allocated.

The amparms are specified as character strings containing one or more specifications of the form ***amparm=value***, separated by commas (for example, **``recfm=v,reclen=100''**). You can specify the amparms in any order and in uppercase or lowercase letters. (However, the case of the value for the **eof** and **prompt** amparms is significant.)

There are two sorts of amparms: those that describe how the file will be processed and those that specify how an MVS file will be created when the filename is specified in **dsn** or **tso** style. All amparms are accepted under both MVS and CMS, but their exact interpretation and their defaults differ from system to system, as described in the following section. Inapplicable amparms are ignored rather than rejected whenever reasonable.

The function descriptions for **afopen**, **afreopen**, and **aopen** provide examples of typical amparm usage.

File processing amparms The file processing amparms may be classified into the following four categories:

File Characteristics

recfm=f/v/u	operating system record format
reclen=nnn|x	operating system record length
blksize=nnn	operating system block size
keylen=nnn	VSAM key length requirement
keyoff=nnn	VSAM key offset requirement
org=value	file organization requirement

File Usage

print=yes|no	file destined to be printed
page=nnn	maximum lines per page (with print=yes)
pad=no|null|blank	file padding permitted
trunc=yes|no	effect of output before end of file
grow=yes|no	controls whether new data can be added to a file
order=seq|random	specifies whether records for a file are normally processed in sequential or random order
commit=yes|no	specifies whether modifications to a file should be committed when the file is closed

`dirsrch=value`	used when opening a CMS Shared File System directory to specify the information to be retrieved from the directory

Terminal Options

`eof=string`	end-of-file string
`prompt=string`	terminal input prompt

VSAM Performance Options

`bufnd=nnn`	number of data I/O buffers VSAM is to use
`bufni=nnn`	number of index I/O buffers VSAM is to use
`bufsp=nnn`	maximum number of bytes of storage to be used by VSAM for file data and index I/O buffers
`bufsize=nnn`	size, in bytes, of a DIV window for a linear data set
`bufmax=n`	number of DIV windows for a linear data set

See "Terminal I/O" on page 3-51 for a discussion of the `eof` and `prompt` amparms. See "VSAM-related amparms" on page 3-80 for a discussion of the VSAM Performance amparms.

The default amparms vary greatly between MVS and CMS, so they are described separately for each system.

File characteristics amparms The `recfm`, `reclen`, `blksize`, `keylen`, `keyoff`, and `org` keywords specify the program's expectations for record format, maximum record length, block size, key length, key offset, and file organization. If the file is not compatible with the program's `recfm`, `reclen`, or `blksize` specifications, it is still opened, but a warning message is directed to the standard error file. If the file is not compatible with the program's `keylen`, `keyoff`, or `org` specifications, a diagnostic message is produced, and the open fails.

If the file is being opened for output and the previous file contents are discarded, the file will, if possible, be redefined to match the program's specifications, even if these are incompatible with the previous file attributes. This is not done if any of the file's contents are to be preserved, because changing the file characteristics may make this data unreadable. (One effect of this is that the characteristics of an MVS partitioned data set are never changed, because even if one member is completely rewritten, other members continue to exist.)

The effects of these amparms are sometimes different from similar specifications on a DD statement, a TSO ALLOCATE command, or a CMS FILEDEF. JCL or command specifications always override any previously established file characteristics, but amparms override only if the library can determine that this is safe.

Details of the file characteristics amparms include the following:

□ The `recfm` amparm defines the file's expected record format. `recfm=f` indicates fixed length records; `recfm=v` and `recfm=u` indicate varying length records. Under MVS, `recfm=v` and `recfm=u` request the DCB attributes RECFM=V and RECFM=U, respectively. Under CMS, the two are equivalent, except when a filemode 4 or OS data set is processed.

  VSAM files are always treated by the library as RECFM V, because they are never restricted by the system to a single record length.

  The `recfm` amparm must be specified as exactly `f`, `v`, or `u`. The inclusion of other characters valid in a JCL specification (for example, `recfm=vba`) is not permitted.

- The **reclen** amparm defines the maximum length record the program expects to read or write. The specification **reclen=x** (which is not permitted with **recfm** specifications other than **v**) indicates that there is no maximum record length.

  Under MVS, the value of **reclen** might not be the same as the LRECL of the data set being opened. For RECFM=V data sets, the LRECL includes 4 bytes of control information, but the **reclen** value contains only the length of the data portion of a record. This allows a **reclen** specification to have the same meaning under MVS and CMS, despite the different definitions of LRECL in the two systems.

  Under MVS, a **reclen=x** output file is created with RECFM=VBS,LRECL=X, which allows arbitrarily long records. Under CMS, a **reclen=x** output allows records up to 65,535 bytes, which is the maximum permitted by CMS.

  For VSAM ESDS and RRDS files, the value of **reclen** must take into account the four-byte key field maintained by the library at the start of the records processed by the program. For example, if the maximum physical record for an ESDS data set is 400 bytes, then you should specify **reclen=404** in the amparms.
- The **blksize** amparm specifies the maximum block size for the file as defined by the operating system. Under MVS and CMS for filemode 4 files, this is equivalent to the DCB BLKSIZE parameter. (Thus, for files with record format V, the V-format control bytes are included in the **blksize** value.)

  For an OpenEdition HFS file, the **blksize** amparm controls the size of the buffer used by the library to access the file. For these files, this is the only effect of specifying **blksize**.

  If a CMS disk file is opened for read-only with the **''seq''** access method and RECFM=F, the **blksize** amparm specifies the library's internal buffer size. If the buffer size is larger than the LRECL of the file, each input operation performed by the library reads as many records as will fit into the buffer.

  When the **''rel''** access method is used to open CMS files, the library transfers data in the units specified. (For example, if you specify **blksize=10000**, the library reads or writes data 10,000 characters at a time.) Under either MVS or CMS, a large **blksize** specification improves performance at the cost of additional memory for buffers.

  The **blksize** amparm under MVS is also used during allocation of a new data set specified with a **dsn**- or **tso**-style filename.
- The **keylen** amparm specifies the length of the key field for a file accessed as keyed. You can also specify **keylen=0** for a file that is not accessed as keyed. For ESDS or RRDS files, if you specify **keylen**, the length must be 4. If the specified length and the actual length do not agree, the open will fail.

  If you open a KSDS that does not already exist, you must specify the **keylen** amparm to correctly create the file. If you access a VSAM file through standard I/O (that is, using text or binary access), **keylen** must either not be specified or be specified as 0.
- The **keyoff** amparm specifies the offset of the key field in the record for a file accessed as keyed. If **keylen=0** is specified, any **keyoff=** specification is ignored. For ESDS or RRDS data sets, you must specify the offset as 0. If you open a VSAM data set that does not already exist and no **keyoff** is specified, then **keyoff=0** is assumed.
- The **org** amparm enables the program to specify a requirement for a particular file organization. For an existing file, the library validates that the file requested has the correct organization. For a new file, the library creates the file with the requested organization, if possible.

The following values are permitted for the **`org`** amparm:

`ps`	is the value specified if the file is an ordinary sequential file, such as an MVS sequential data set, a CMS disk file, a tape file, or a CMS spool file. To ready the directory, specify the value **`ps`** for a file that is a PDS.
`os`	is the value specified if the file is an OS format file under CMS, such as a filemode 4 file or a file on an OS disk.
`po`	is the value specified if the file is a partitioned data set, or a CMS MACLIB or TXTLIB. Under systems supporting PDSEs, the file can be either a regular PDS or a PDSE.
`pds`	is the value specified if the file is a regular (non-PDSE) PDS.
`pdse`	is the value specified if the file is a PDSE.
`ks`	is the value specified if the file is a VSAM KSDS.
`es`	is the value specified if the file is a VSAM ESDS.
`rr`	is the value specified if the file is a VSAM RRDS.
`ls`	is the value specified if the file is a VSAM linear data set.
`byte`	is the value specified if the file is an OpenEdition HFS file.

Certain **`org`** values are treated as equivalents in some systems to permit programs to be ported from one environment to another. Notably, the **`org`** values **`pds`** and **`pdse`** are treated like the value **`po`** under systems not supporting PDSEs, and the values **`os`** and **`ps`** are treated synonymously under MVS.

File usage amparms File usage amparms allow the program to specify how a file will be used. A specification that cannot be honored may cause the open to fail, generate a warning message, or cause a failure later in execution, depending on the circumstances. The exact treatment of these amparms is highly system-dependent.

- The amparm **`print=yes`** or **`print=no`** indicates whether the file is destined to be printed. If you specify **`print=yes`**, ANSI carriage control characters are written to the first column of each record of the file to effect page formatting, if the file format permits this. In your C program, you can write the **`'\f'`** character to go to a new page and the **`'\r'`** character to perform overprinting.

  `print=yes` is allowed only for files that are accessed as a text stream and whose open mode is **`'w'`** or **`'a'`**. If these conditions are satisfied but the file characteristics do not support page formatting, a warning message is generated, and no page formatting occurs.

  If you specify **`print=no`**, then the **`'\f'`** and **`'\r'`** characters in output data are treated as normal characters, even if the file characteristics will permit page formatting to occur.
- The amparm **`page=`*`nnn`*** specifies the maximum number of lines that will be printed on a page. It is meaningful only for files opened with **`print=yes`**, or for which **`print=yes`** is the default. It is ignored if specified for any other file.
- The amparm **`pad`** specifies how file padding is to be performed. **`pad=blank`** requests padding with blanks, **`pad=null`** requests padding with null characters, and **`pad=no`** requests that no padding be performed. If **`pad=no`** is specified, a record that requires padding is not written and a diagnostic message is generated.

  The **`pad`** amparm is meaningful only for files with fixed-length records. For files accessed as text, pad characters are added as necessary to each output record and removed from the end of each input record. For output files accessed as binary, padding only applies to the last record, and for input files accessed as binary, padding is never performed.

- When new data is written to a file before the end of file, the amparm `trunc` specifies whether existing records following the current file position are to be erased or preserved. `trunc=yes` specifies erasure; `trunc=no` specifies preservation. If the `trunc` specification cannot be honored, the open fails. The primary use for this parameter is to indicate a program dependency on truncation or nontruncation and thereby avoid inappropriate file updates.

  You need to use the `trunc` amparm only when you use open modes `'r+'`, `'r+b'`, `'w'`, `'w+'`, or `'w+b'` and one of the file positioning functions (`fseek`, `rewind`, or `fsetpos`) to position before the end of file. Do not specify the `trunc` amparm with UNIX style binary I/O; UNIX style binary I/O always exhibits `trunc=no` behavior. You can change the length of a record in a file only if it is the last record, or if `trunc=yes` is in effect.
- The amparm `grow=yes` or `grow=no` controls whether new data can be added to a file. The default is always `grow=yes`, which permits the addition of new records. The `no` specification is only permitted when the open mode is `'r+'`, `'r+b'`, or `'r+k'`, and when `trunc=yes` is not specified. When `grow=no` is specified, attempts to add new records to the file will fail. For some file types, notably MVS PDS members, use of `grow=no` can lead to performance improvements. In particular, for PDSE members, the `fseek` and `fsetpos` functions are supported if you specify `grow=no`, but not with `grow=yes`.
- The amparm `order=seq` or `order=random` specifies whether records for the file are normally processed in sequential or random order. This is specified as `order=seq` for sequential order, or `order=random` for random order. This amparm is meaningful for VSAM files with keyed access and for CMS shared files; correct specification can lead to performance improvements. For all other file types, this amparm is ignored. The default is determined by the access method. For VSAM, the default is `order=random`; for CMS shared files, the default is selected by CMS.
- The amparm `commit=yes` or `commit=no` specifies whether modifications to the file should be committed when the file is closed. The `no` specification for the `commit` amparm is supported only for CMS Shared File System files, and this specification is rejected for any other file type. When `commit=no` is in effect, you must call the `afflush` function to commit updates to a shared file. If you close without calling `afflush`, the updates are rolled back, and the file is left unchanged.

  When `commit=no` is specified in a call to `aopen` (using UNIX style I/O), the behavior is slightly different. See the `fsync` function description for a discussion of this case.
- The `dirsrch` amparm opens a CMS Shared File System directory to specify the information to be retrieved from the directory.

Amparms - MVS details

This section discusses amparms under MVS. It provides an explanation of and defaults for each amparm.

File characteristics For input files, or output files in which some or all of a file's previous contents are preserved, the file characteristics amparms serve as advice to the library regarding the file characteristics expected. If the actual file does not match the program's assumptions, a warning message is generated. In some cases, no warning is generated, even though the file characteristics are not exactly what the program specified. For instance, if a program specifies amparms of `''recfm=v,reclen=80''` and opens an input file with LRECL 40, no diagnostic is generated, because all records of the input file are consistent with the program's assumption of input records with a maximum length of 80 bytes.

To determine the characteristics of a file, the library merges information from the amparms, control language specifications, and the data set label. Unlike amparms information, control language specifications always override the physical file characteristics recorded in the label.

For each of these amparms, processing is as follows:

`recfm` If you specify **`recfm=f`**, the program expects records of equal length, and a warning is generated if the file does not have fixed-length records (blocked or unblocked).

If **`recfm=v`** or **`recfm=u`** is specified for a read-only file, no diagnostic is ever generated. For a write-only or update file, a warning is generated if the MVS RECFM does not match the amparm.

VSAM linear and RRDS files are always considered to have RECFM F, and other VSAM files and OpenEdition HFS files are considered to have RECFM V.

`reclen` If you specify **`reclen=`*nnn*** for a read-only file, a warning is generated if the file's record size is larger, or if it is not equal and the record format is fixed. If **`reclen=x`** is specified for a read-only file, a diagnostic is never generated, except when the record format is fixed. Note that under MVS, the program's **`reclen`** specification is compared to the LRECL−4 for a V-format file, not to the LRECL itself. (Additionally, for a file with carriage control characters, the control character is not counted.)

If you specify **`reclen=`*nnn*** for a write-only or update file, a warning is generated if the file's record size is not the same as the **`reclen`** specification. If you specify **`reclen=x`**, a warning is generated unless the file has RECFM=VBS or RECFM=VS and LRECL=X.

VSAM linear data sets are always considered to have **`reclen=4096`**.

`blksize` If you specify **`blksize=`*nnn***, a warning is generated only if the actual blksize is greater than that specified.

Note: When a write-only or update file is opened and none of the file's previous contents are preserved, the file's characteristics are changed to correspond to the program's amparms specifications. The details of this process are outlined here:

- **`recfm`** specifications of **`f`**, **`v`**, and **`u`** become MVS RECFM's of FB, VB, and U, respectively. However, if you select the **`''rel''`** access method, RECFM FBS is chosen. If the **`reclen`** is **`x`** or the **`blksize`** is less than the **`reclen`** with **`recfm=v`**, MVS RECFM VBS is requested. Finally, if you also specify **`print=yes`**, ANSI carriage control characters (RECFM=A) are added.
- A **`reclen=x`** specification requests use of LRECL=X. **`reclen=`*nnn*** requests a LRECL of ***nnn***, unless the record format includes V, in which case, it requests ***nnn***+4.
- A **`blksize=`*nnn*** specification requests an MVS BLKSIZE of ***nnn***. If the requested BLKSIZE is not compatible with the chosen RECFM and LRECL, the BLKSIZE is rounded, if possible.

When a file is opened and neither the file nor any amparms fully specify the file characteristics, the following rules apply:

- If the **`''rel''`** access method is in use, RECFM=F, FS, or FBS is required. The default record length is 1, and the default blksize is 4080.
- For the **`''seq''`** access method, choices are made based on device type and the presence or absence of a **`print`** specification, as shown in Table 3.4.

Table 3.4
MVS Default File Characteristics Amparms

Device Type	`recfm`	`reclen`	`blksize`
Card punch	`f`	80	80
Printer/SYSOUT/DUMMY	`v`	132	141
Other (print=yes)	`v`	132	6144
Other (print=no)	`v`	255	6144

File usage The amparm `print` has two uses: it specifies whether the corresponding file includes ANSI carriage control characters, and it specifies whether the C library should process the control characters `'\r'` and `'\f'` to effect page formatting. If `print=no` is specified, then `'\r'` and `'\f'` are simply treated as data characters, even if the file can support page formatting. If you specify `print=yes`, then the library attempts to perform page formatting. However, if the associated file does not have the correct attributes, `'\r'` and `'\f'` are treated as new lines, and a warning message is generated when the file is opened. If neither `print=no` nor `print=yes` is specified, the library chooses a default based on the attributes of the external file. However, `print=yes` is supported only for a file accessed as a text stream.

Under MVS, a file is considered to be suitable for page formatting if it has the following characteristics:

- It is a SYSOUT or printer file.
- Its RECFM includes the letter A, indicating support for ANSI control characters.

For files with RECFM A, space for the control character is included in the LRECL, but not in any `reclen` specification made by the program.

The `page` amparm, which specifies the number of lines to be printed on each page of a `print=yes` file, does not have a default. That is, if `page` is not specified, page ejects will occur only when a `'\f'` is written to the file.

Note: The `print` and `page` amparms are ignored when opening files in the OpenEdition hierarchical file system. Control characters are always written to an HFS file, regardless of whether you specified `print`.

The `pad` amparm specifies whether padding of records will take place, and, if so, whether a blank or a null character (`'\0'`) will be used as the pad character. The default depends on the library access method and the open mode, as follows:

- If the access method is `''rel''`, the default is `pad=null`.
- If the access method is `''seq''` and the file is accessed as text, the default is `pad=blank`.
- If the access method is `''seq''` and the file is accessed as binary, the default is `pad=no`; that is, padding is not performed.

The `trunc` amparm indicates to the library whether the program requires that output before the end of file erase the following records or leave them unchanged. Under MVS, whether existing data are preserved in this situation depends only on file type and access method. If `trunc` is omitted, the value corresponding to the file's actual behavior is used; if a conflicting `trunc` specification is made, the file fails to open. If a file is processed by the `''rel''` access method, is a VSAM file, or is in the OpenEdition HFS, only `trunc=no` is supported. For all other MVS file types, only `trunc=yes` is supported.

The `grow` amparm indicates to the library whether new data can be added to a file opened for `''r''` or `''r+''`. The specification `grow=no` informs the library that the program will only replace existing records of a file, rather than adding any data to the end. When you specify `grow=no` for a file processed with BSAM, the library can open it for UPDAT rather than OUTIN. This allows the library to support use of the `fseek` or `fsetpos` functions on a PDSE member. `grow=no` implies `trunc=no`.

File creation amparms These amparms are used under MVS with filenames specified in `dsn` or `tso` style when the file does not exist and must be created. These amparms are accepted under CMS, and for `ddn`-style names or existing files under MVS, but in these cases they are ignored.

Note: VSAM files can be created directly by a C program only in MVS, and only if the Storage Management Subsystem (SMS) is active. On CMS, or if SMS is not available, VSAM files must be created by the Access Method Services (AMS) utility before they can be accessed by a C program.

The file creation amparms are as follows:

`alcunit=block|trk|cyl|rec|krec|mrec`	unit of space allocation
`space=`*nnn*	primary amount of space to allocate
`extend=`*nnn*	secondary amount of space to allocate
`dir=`*nnn*	number of PDS directory blocks
`vol=volser`	requested volume serial number
`unit=name`	requested unit name.
`rlse=yes|no`	release unused file space when file is closed
`dataclas=name`	data class for a new file
`storclas=name`	storage class for a new file
`mgmtclas=name`	management class for a new file

The meanings of these amparms and their defaults are discussed in the following list. Default values are site-specific and may have been changed at the time the SAS/C library was installed. Consult your SAS Software Representative for C compiler products to determine the defaults at your site.

- The `alcunit` amparm defines how the values specified for `space` and `extend` are to be interpreted. If you specify `alcunit=block`, the `space` and `extend` values are interpreted as the number of physical blocks to allocate. (If you specify `alcunit=block`, you must also specify the `blksize` amparm to define the size of the blocks.) Similarly, `alcunit=trk` specifies that the `space` and `extend` values are to be interpreted as the number of disk tracks, and `alcunit=cyl` specifies that they are to be interpreted as the number of disk cylinders.

  If you use the `rec` specification, the `space` and `extend` amparms are expressed in numbers of records. The `krec` specification expresses these values in units of 1024 records, and the `mrec` specification expresses these values in units of 1,048,576 records. If you use one of these specifications, you must also specify the `reclen` amparm to define the record length. Use of these options is recommended only when the Storage Management Subsystem of MVS is installed and active. If SMS is not active and either `rec`, `krec`, or `mrec` is specified for the `alcunit` amparm, the library attempts to convert the specification to an equivalent specification in blocks, tracks, or cylinders. However, the conversion is at best approximate, and may allocate substantially more or less space than actually required.
- The `space` amparm specifies the amount of disk space to be initially allocated to the file, in the units specified by `alcunit`. For instance, the amparm string `''alcunit=block,blksize=5000,space=100"` requests enough space to hold 100 blocks of 5000 characters each. If there is not enough disk space available, the open fails.

- The **extend** amparm specifies the amount of additional disk space allocated to a file if the existing space runs out during processing. A file can be extended up to 15 times, after which any attempt to add more data to the file fails. As with **space**, the **extend** value is interpreted in the units specified by **alcunit**.

  The amparms **alcunit**, **space**, and **extend** must be specified together. Specifying **space** without **alcunit** or **extend** without **space** will cause the open to fail without creating a file.

  The amparm specification **''alcunit=alc,space=spc,extend=ext"** is equivalent to the JCL specification SPACE=(alc,(spc,ext)), where an "alc" of "block" is replaced by the value of the **blksize** amparm, and where an "alc" of "rec", "krec" or "mrec" is replaced by an appropriate AVGREC JCL parameter.

  When a nonexisting data set specified in **dsn** or **tso** style is opened and no space specification is supplied by the program, a default amount of space is allocated. The standard default allocation is that specified by **"alcunit=block,blksize=1000, space=10,extend=3"**.
- The **dir** amparm specifies the number of directory blocks to allocate when a partitioned data set is created. (See *MVS/DFP Using Data Sets* for more information on PDS directories.) The **dir** amparm applies only to partitioned data sets. If **dir** is specified in the amparms but the filename does not include a member name, a member name of TEMPNAME is assumed. If no **dir** amparm is specified and the filename does not include a member name, a sequential data set is created. If a member name is specified, a default **dir** value of 5 is assumed.

  You do not have to specify the **dir** amparm if you request creation of a PDSE using the **org=pdse** amparm, since pre-allocation of directory blocks is not required.
- The **vol** amparm specifies the volume serial on which a new data set will be created. If no **vol** amparm is specified for a new data set, the system is allowed to select the volume on which to create the data set.
- The **unit** amparm specifies a unit name (such as SYSDA) to use when allocating a new file. This has the same effect as the JCL UNIT keyword. See the IBM publication *MVS/XA ESA JCL Reference*, for more information. If **unit** is not specified, the normal procedures at your site for unit selection are used. (For instance, for TSO users, the default unit name is defined as part of the user profile.)
- **rlse** specifies whether unused file space should be released when the file is closed. This option only has an effect when the file is created at the time it is opened. The default is **rlse=no**. Use of **rlse=yes** with a file that is opened and closed repeatedly may cause the maximum file size to be smaller than if **rlse=no** had been specified, due to repeated release of temporarily unused space. You should not specify the **rlse** amparm for VSAM files.
- **dataclas** specifies the data class for a new file. This amparm is ignored if SMS is not active, or if the file already exists. Your site may choose to ignore this specification.
- **storclas** specifies the storage class for a new file. This amparm is ignored if SMS is not active, or if the file already exists. Your site may choose to ignore this specification.
- **mgmtclas** specifies the management class for a new file. This amparm is ignored if SMS is not active or if the file already exists. Your site may choose to ignore this specification.

See the IBM publication *MVS/DFP Storage Administration Reference* for further information on SMS concepts such as data class and storage class.

Amparms - CMS details

This section discusses amparms under CMS. It provides an explanation of each amparm, including default values.

File characteristics For input files, or output files in which some or all of a file's previous contents are preserved, the file characteristics amparms serve as advice to the library about the file characteristics expected. If the actual file does not match the program's assumptions, a warning message is generated. To determine the characteristics of a file, the library tries to merge information from the amparms, the FILEDEF options (for a `ddn`-style filename), and from an existing file with the same fileid.

The processing for each of the file characteristics amparms is described here. Unless otherwise specified, the description is for CMS disk files.

`recfm` If `recfm=f`, the program expects records of equal length. If the file is not a RECFM F file, a warning message is generated. Similarly, if `recfm=v`, the program expects records of varying length. If the file is not a RECFM V file and the file is opened for write-only or for update, a warning message is generated. `recfm=u` is treated as if it were `recfm=v`.

`reclen` For files with fixed length records, `reclen` specifies the length of the records. If the record length specified by `reclen` does not match the LRECL of the file, a warning message is generated. `reclen=x` cannot be used with fixed format files.

If the file has varying length records, `reclen` specifies the maximum length of the records. If the LRECL of the file exceeds that specified by `reclen`, a warning message is generated. `reclen=x` implies that the records may be of any length up to 65,535.

`blksize` The `blksize` amparm is used with the `''rel''` access method to specify the internal buffer size used by the library, which in turn specifies the number of records read or written by the library in each I/O operation. The `blksize` for the file should be a multiple of the file's logical record length. If it is not, it is rounded to the next higher multiple.

For files with fixed-length records opened for read only, `blksize` can also be used with the `''seq''` access method. In this case, `blksize` specifies the library's internal buffer size. If the buffer size is larger than the LRECL of the file, each input operation performed by the library reads as many records as can fit into the buffer. For example, if the file has 80-character records, specifying `blksize=4000` causes the library to read 50 records at each input operation.

When an existing write-only or update file is opened, and none of the file's previous contents are preserved, the old file is erased and a new file is created. The characteristics of the new file are those specified by the amparms.

`recfm` `recfm=f` and `recfm=v` cause the file to be created as RECFM F or RECFM V, respectively. Again, `recfm=u` is treated as if it were `recfm=v`.

`reclen` specifies the maximum (and minimum, for `recfm=f`) logical record length for the file. `reclen=x` indicates that the records may be of any length up to CMS's maximum of 65,535.

`blksize` specifies the buffer size used by the library when performing I/O operations on the file.

If the file characteristics are not completely described by the amparms, the FILEDEF options (when the `ddn`-style filename is used), or the file, the following defaults apply:

- □ If the `''rel''` access method is in use, the amparms `recfm=f`, `reclen=1`, `blksize=4080` are assumed.
- □ For the `''seq''` access method, choices are based on the virtual device type, filetype, and the presence of the `print` amparm.
- □ For files with filetype LISTING, files written to the virtual printer, or files with `print=yes` specified, the defaults are `recfm=v` and `reclen=132`.
- □ If the file is written to the virtual punch, the defaults are `recfm=f` and `reclen=80`.
- □ For other files, if the `recfm` amparm is not specified, the default is `''v''`. If the `recfm` is `''f''`, the default `reclen` is 80; otherwise, the default is `reclen=x`. For files in OS-format or for tape files, amparms are processed as described in "Amparms - MVS details" on page 3-40. The single exception is that the default `blksize` for tape files is 3600, rather than 6144.

File usage The amparm `print` has two uses: it specifies whether the corresponding file includes ANSI carriage control characters, and it specifies whether the C library should process the control characters `'\r'` and `'\f'` to effect page formatting. If `print=no` is specified, then `'\r'` and `'\f'` are simply treated as data characters, even if the file can support page formatting. If you specify `print=yes`, then the library attempts to perform page formatting, but if the associated file does not have the correct attributes, `'\r'` and `'\f'` are treated as new lines. (A warning message is generated when the file is opened in this case.) If neither `print=no` nor `print=yes` is specified, the library chooses a default based on the attributes of the external file. However, `print=yes` is supported only for a file accessed as a text stream.

Under CMS, any disk file may be used with page formatting. If the filetype of the file is LISTING or the file is written to the virtual printer, the file is assumed to require ANSI control characters in the first byte of each record. (If a disk file has control characters in byte 1 and does not have a filetype of LISTING, the CMS command PRINT prints the file incorrectly unless the CC option is used.)

The `page` amparm, which specifies the number of lines to be printed on each page of a `print=yes` file, does not have a default. That is, if you do not specify `page`, page ejects will occur only when a `'\f'` is written to the file.

The `pad` amparm specifies whether padding of records will take place, and, if so, whether a blank or a null character (`'\0'`) will be used as the pad character. The default depends on the library access method and the open mode, as follows:

- □ If the access method is `''rel''`, the default is `pad=null`.
- □ If the access method is `''seq''` and the file is accessed as text, the default is `pad=blank`.
- □ If the access method is `''seq''` and the file is accessed as binary, the default is `pad=no`; that is, padding is not performed.

The `trunc` amparm indicates to the library whether the program requires that output before the end of file erase following data or leave them unchanged. CMS disk files support both `trunc=yes` and `trunc=no`. By default, `trunc=no` is assumed; that is, data are not erased following a modified record. Shared file system files support only `trunc=no`. All other types of files support only `trunc=yes`, except for VSAM, which supports only `trunc=no`. If a program's `trunc` specification is not supported for the file being opened, the open fails, and a diagnostic message is generated.

File Positioning

As described in "Technical Background" on page 3-2 , the 370 operating systems provide a relatively inhospitable environment for the standard C file positioning functions. For this reason, you should read this section carefully if your application makes heavy or sophisticated use of file positioning. Some understanding of MVS or CMS I/O internals is helpful.

The details of file positioning depend heavily on the I/O package and library access method used, the stream type (text or binary), and the file organization and attributes. The following discussion is organized primarily by I/O package.

File positioning with UNIX style I/O

When UNIX style I/O is used to access a file as binary, file positioning is fully supported with the `lseek` function, which can be used to seek to an arbitrary location in the file. However, when UNIX style I/O is used to access a file as text, the seek address is interpreted in an implementation-specific way, as when the `fseek` function is used. This means that, for a text stream, you should use `lseek` only to seek to the beginning or end of a file, or to a position returned by a previous call to `lseek`.

The `lseek` function accepts three arguments: a file number, an offset value, and a symbolic value indicating the starting point for positioning (called the seek type). The seek type is interpreted as follows:

- If the seek type is `SEEK_SET`, the offset is interpreted as an offset in bytes from the start of the file.
- If the seek type is `SEEK_CUR`, the offset is interpreted as the offset in bytes from the current position in the file.
- If the seek type is `SEEK_END`, the offset is interpreted as the offset in bytes from the end of file.

If the seek type is `SEEK_CUR` or `SEEK_END`, the offset value may be either positive or negative. `lseek` can be used with a seek type of `SEEK_SET` and an offset of 0 to position to the start of a file, and with a seek type of `SEEK_END` and an offset of 0 to position to the end of a file.

Positioning beyond the end of a binary file with `lseek` is fully supported. Note that positioning beyond the end of a file does not in itself change the length of the file; you must write one or more characters to do this. When you write data to a position beyond the end of file, any unwritten positions between the old end of file and current position are filled with null characters (`'\0'`).

The `lseek` function returns the current file position, expressed as the number of bytes from the start of the file. It is frequently convenient to call `lseek` with a seek type of `SEEK_CUR` and an offset of 0 to obtain the current file position without changing it.

Recall that, except for files suitable for `''rel''` access and OpenEdition HFS files, using UNIX style I/O is relatively inefficient. See "Choosing I/O Techniques and File Organization" on page 3-21 for more information on the advantages and disadvantages of using UNIX style I/O.

File positioning with standard I/O (fgetpos and fsetpos)

Standard I/O supports the `fgetpos` and `fsetpos` functions to obtain or modify the current file position with either a text or a binary stream. Both `fsetpos` and `fgetpos` accept two arguments: a pointer to a `FILE` object and a pointer to an object of type `fpos_t`. For `fsetpos`, the `fpos_t` object specifies the new file position, and for `fgetpos`, the current file position is stored in this object. The exact definition of the `fpos_t` type is not specified by the ISO/ANSI C standard and, if you intend your program to be portable, you should make no assumptions about it. However, an understanding of its implementation by the SAS/C library can be useful for debugging or for writing nonportable applications.

The library defines **fpos_t** using the following **typedef**:

```
typedef struct {
   unsigned long _recaddr;    /* hardware "block" address */
   long _offset;              /* byte offset within block */
} fpos_t;
```

The first element of the structure (**_recaddr**) contains the address of the current block or record. The exact format of this value is system- and filetype-dependent. The second element of the structure (**_offset**) contains the offset of the current character from the start of the record or block addressed by **_recaddr**. In some cases, this offset may include space for control characters.

A more precise definition of these fields for commonly used file types follows.

- For a CMS disk file processed by the **''seq''** access method, the **_recaddr** is the current record number, and the **_offset** value is the offset of the current character from the start of the record. Record numbering starts at 1, not 0.
- For a VSAM ESDS, the **_recaddr** is the RBA (relative byte address) of the current record, and the **_offset** value is the offset of the current character from the start of the record.
- For an MVS disk file processed by the **''seq''** access method, the **_recaddr** is the TTR of the block preceding the block containing the current record. If the file is a PDS member, the TTR is computed relative to the start of the member, not the start of the PDS. The **_offset** value is the offset of the current character from the start of the block containing the current record. In computing the offset, each record is treated as if it were terminated with a single new-line character, even for a file accessed as a binary stream. This technique allows the library to easily distinguish the end of a record from the start of the next record.
- For files processed by the **''rel''** access method and OpenEdition HFS files, the exact values in **_recaddr** and **_offset** depend on the file attributes and on previous processing. You should use **fseek** and **ftell** rather than **fsetpos** and **fgetpos** for these files if you need to construct file positions manually.

fsetpos and **fgetpos** are implemented to be natural and efficient, and not to circumvent limitations or peculiarities of the operating systems. For this reason, you should be aware of the following:

- In special cases in which the operating system or I/O device does not provide adequate support, **fsetpos** and **fgetpos** may fail. These cases are outlined in Tables 3.5 and 3.6. When **fsetpos** or **fgetpos** cannot be supported, a diagnostic message is generated, and the return value from the function indicates that an error occurred. A proper call to **fsetpos** will never indicate success when the requested operation is not supported by the file.
- The effect of seeking past the end of the file or past the end of a record is undefined. The file type and the situation determine whether this error will be detected. Because **fgetpos** never returns a file position of this sort, this situation can arise only if your program manufactures its own **fpos_t** values. Seeking past the end of file is supported for files processed by the **''rel''** access method. For these files, writing a character beyond the end of file causes intervening positions to contain hexadecimal 0s.
- Positioning using an invalid **_recaddr** value is frequently undetected, or causes an error only when the file is next read or written.
- Writing after seeking to a position before the end of file may cause any following data to be discarded, depending on the file type and whether the **trunc** amparm is specified when the file is opened.

- You should not compare **fpos_t** values. In some cases, especially for files with spanned records, several different values may identify the same location in the file.
- Because of the use of TTRs under MVS, file positions may differ between copies of the same file. Similarly, VSAM RBAs may change if the control interval or control area size is changed.

File positioning with standard I/O (fseek and ftell)

In many cases, when you process a file with standard I/O, you can use the **fseek** and **ftell** functions to obtain or modify the current file position. Because **fsetpos** and **fgetpos** are relatively new additions to the standard C language, **fseek** and **ftell** are more portable. However, they are also more restricted in their use. Full **fseek** and **ftell** functionality is available only when you use the **''rel''** access method, when a file is accessed as a text stream, or when the file is in the OpenEdition hierarchical file system. For files processed with the **''rel''** access method and for HFS files, **fseek** and **ftell** function exactly like **lseek** does for UNIX style files.

The **fseek** function accepts three arguments: a pointer to a **FILE** object, an offset value, and a symbolic value indicating the starting point for positioning (called the seek type). The offset value is a **long** integer, whose meaning depends on whether the file is accessed as text or binary. For binary access, the offset value is a number of bytes. For text access, the offset value is an encoded file position whose interpretation is unspecified. The seek type is one of the values **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, indicating positioning relative to the start of file, the current position, or the end of file, respectively.

When you access a file as binary, the **fseek** offset value is simply interpreted as an offset from the point specified by the seek type. For instance, **fseek(f, -50L, SEEK_CUR)** requests positioning 50 characters before the current character.

Note: Because the **fseek** offset value has type **signed long**, only files whose size is less than 2**31 bytes can be supported in a portable fashion. However, for files accessed using the **''rel''** access method, or stored in the OpenEdition HFS, the offset value is interpreted as **unsigned long**, thus allowing the use of files whose size is less than 2**32-1.

When you access a file as text, only certain combinations of offset and seek type are meaningful. When the seek type is **SEEK_CUR** or **SEEK_END**, only an offset value of 0 is meaningful, requesting no change in positioning or positioning to the end of file, respectively. When the seek type is **SEEK_SET**, any valid file position previously returned by **ftell** is accepted as an offset. Additionally, you can use an offset of 0 with **SEEK_SET** to reposition to the start of the file.

The **ftell** function accepts one argument, a pointer to a **FILE** object, and returns a **long** integer defining the current file position. For a file accessed as binary, the returned file position is the number of bytes from the start of the file. For a file accessed as text, the returned file position is in an internal format.

Note: For a text stream, **ftell** is the only safe mechanism for obtaining a file position for later use by **fseek**; that is, you cannot construct meaningful file positions in another way.

ftell computes the encoded file position for a text stream by first calling **fgetpos** to obtain the file position in **fpos_t** format. (Thus, if **fgetpos** is not usable with a file, neither is **ftell**.) Then, initial portions of the **_recaddr** and **_offset** fields of the result are combined in a manner that depends on the file organization. Because the size of an **fpos_t** value is 8 bytes and the size of a **long** integer is only 4 bytes, information is lost if either the **_recaddr** value or the **_offset** value is too large. The library detects this loss of information and returns an error indication, rather than an incorrect file position. For specific file types, the conditions under which a file position cannot be successfully returned are listed here:

- □ an MVS disk file or PDS member larger than 256 tracks
- □ a CMS-format disk file containing more than 65,535 records
- □ a tape file containing more than 65,535 blocks
- □ a VSAM file containing more than 16,777,215 characters, or a record longer than 255 bytes.

Even for files that exceed one of these limits, `ftell` returns an error only when the actual file position is outside the limits.

When used with a binary stream, full `fseek` functionality is restricted to the `''rel''` access method, but note that `fseek` with a 0 offset value is usually supported. This means that you can use `fseek` with almost any file to rewind, position to end of file, or switch between reading and writing. The exceptions are noted in Tables 3.5 and 3.6.

Table 3.5
MVS Files with Restricted Positioning

File Type	Restrictions
terminal	positioning not allowed
card reader	positioning not allowed
OpenEdition pipe	positioning not allowed
DD */DATA	only `fseek` with a 0 offset supported
printer/card punch/SYSOUT	rewind and seek to the end of file accepted but ignored
keyed VSAM (KSDS)	only `fseek` with a 0 offset supported
PDS member	seek to the end of file not supported; switch between reading and writing only supported at the start of file unless `grow=no`
PDSE member	seek to the end-of-file not supported; seeking other than rewind only allowed if read-only or `grow=no`
`RECFM=FBS` (accessed as text)	seek to the end of file not supported
multivolume disk or tape file	only rewind supported if not opened for append; only seek to the end of file supported if opened for append
unlabeled tape with `DISP=MOD`	only seek to the end of file supported
concatenated sequential files	only `fseek` with a 0 offset supported

Table 3.6
CMS Files with Restricted Positioning

File Type	Restrictions
terminal	positioning not allowed
reader/printer/punch	rewind and seek to the end of file accepted but ignored
keyed VSAM (KSDS)	only `fseek` with a 0 offset supported
MVS PDS member	positioning not allowed with %MACLIB or %TXTLIB filename used

Note: The warnings given in the previous section for use of `fsetpos` are equally applicable to `fseek`.

Terminal I/O

Performing I/O on an interactive device such as a TSO or CMS terminal is quite different from performing I/O on a disk or tape file. Some programs need to use terminal and nonterminal files interchangeably, while others need to take advantage of the special properties of terminal files. Some specific differences between terminal I/O and other I/O follow.

Note: The following considerations apply when you read from or write to the TSO or CMS terminal. They do not necessarily apply when you read or write from an OpenEdition terminal under the OpenEdition shell. The behavior of an OpenEdition terminal is defined by the POSIX standards. See the POSIX 1003.1 standard for further information.

- A terminal file can be opened only with open mode `'r'`, `'w'`, or `'a'`. Open for append and write-only are treated identically. If you want to both read from and write to the terminal, you must use two files.
- The distinction between text access and binary access for a terminal file is quite different from the distinction for other file types. Binary access should be used only when it is necessary to send or receive terminal control characters. (See "Text access and binary access" on page 3-52.)
- In theory, a terminal file has no defined end. In practice, many programs require a way that the end of file can be signalled from the terminal. When a terminal file is accessed as text, the string `''EOF''` (uppercase letters required) is normally interpreted as the end of file. (This string may be changed by specifying the `eof` amparm when the file is opened.) When a terminal file is accessed as binary, a null input line will be treated as end of file.
- You must use the `''term''` access method for terminal I/O.
- When UNIX style I/O is used to process a terminal file, the file is always accessed as text.

Buffering, flushing, and prompting

Most implementations of standard I/O perform buffered I/O; that is, characters are collected in a buffer and transmitted, one block at a time. This can cause problems for I/O to the terminal. For instance, if a terminal output file is buffered, it is possible for a terminal read to be issued before an output message asking the user to enter the input is transmitted. To write correct and portable interactive programs, it is important to understand the different ways that terminal I/O can be implemented. Some of the possible approaches are as follows:

- □ Some implementations require the programmer to use the **fflush** function to force an output buffer to be written. Programs that use this technique are portable, because this works with almost any C library implementation.
- □ Some implementations buffer terminal output on a line-at-a-time basis. This means that as long as a program writes complete lines of output (each one terminated with a new-line character), there is no problem with delayed messages.
- □ Some implementations inhibit buffering for terminal files. This technique avoids the problem, but it is not practical under MVS or CMS.
- □ Some implementations automatically flush terminal output buffers before terminal input is requested. This is the technique used by the SAS/C library. Other implementations apply this technique only to the standard input and output streams (**stdin** and **stdout**). This implementation allows many simple programs to run as intended without solving the general problem.

Another situation that varies from one implementation to another, depending on the buffering strategy, is the effect of writing characters to several terminal files at the same time. In implementations that do not buffer terminal I/O, all the output characters are transmitted in the order that they are written. In an implementation that performs buffering, the output probably consists of complete lines of output, each line associated with a particular file. With the SAS/C library I/O functions, the buffer for one terminal output file is flushed when a character is written to another. This means that characters are transmitted in the same order as they are written and that characters written to two different files do not appear in the same output line.

Automatic prompting

When you use **scanf** or a similar function to read input from the terminal, it can be difficult to write prompts at all points where terminal input is required. For example, the following call reads two integers from the standard input stream (normally the terminal):

```
scanf("%d %d", &i, &j)
```

If only one integer is entered, **scanf** issues another read to obtain the second integer without returning to the program. This means that the program is unable to issue a prompt or message to tell you that more input is required.

When you open a terminal input file, the library allows you to specify a prompt that will be written by the library immediately before a read is issued to the terminal. This allows each input request to be automatically preceded by a prompt. You can also have more than one terminal input file, each with a different prompt, allowing you to easily distinguish one file from another. This feature is requested with an amparm when you use **afopen** or **aopen** to open the file.

Text access and binary access

One of the distinctions between text access and binary access to a nonterminal file is the treatment of control characters. Certain control characters, such as the new-line and form feed characters, may be transformed during text input and output. Binary access is required to transmit control characters without alteration.

The situation is similar for input and output to the terminal. Usually, the terminal is accessed as a text stream. In this mode, the new-line character separates lines, and a form feed issues several blank lines to simulate the effect of a form feed on a printer. Other control characters are removed from input and output. Both TSO and CMS are sensitive to the data sent to the terminal, and incorrect use of control characters can cause the user to be disconnected or logged off. Thus, the removal of control characters is a safety measure that prevents unpleasant consequences when uninitialized data are sent to the terminal.

However, some applications need the ability to send control characters to the terminal. This is supported by accessing the terminal as a binary stream. When a binary stream is used, output data are sent to the terminal without modification, and input data are only minimally edited. The new-line character still separates lines of output, because both TSO and CMS support this use of the new-line character. Note that even though the SAS/C library does not modify data transmitted to and from the terminal when binary access is used, the data can be modified by system software, including the VM control program, VTAM, and communication controller software. Also recall that incorrect data can cause disconnection or other errors, so use this technique with caution.

Terminal I/O under MVS-batch

Programs that open the terminal for output or append can be run in MVS-batch. In this case, the output is written to the file defined by the DDname SYSTERM. The SYSTERM data set must have record format VBA and LRECL 137, but any BLKSIZE is permissible.

You may have several terminal output files open simultaneously. In this case, lines written to the various files are interspersed, as they would be if the program were running interactively. Emulation of terminal input under MVS-batch is not supported.

Terminal I/O amparms

When you use **`afopen`** or **`aopen`** to open a terminal file, you can specify the following amparms in addition to those discussed earlier in “Opening Files” on page 3-27:

`eof=string`	specifies an end-of-file string.
`prompt=string`	specifies a terminal input prompt.

The following amparms are ignored if the file to be opened is not the terminal:

- □ The **`eof`** amparm enables you to specify the string that is interpreted as end-of-file if entered from the terminal. It applies only when the terminal is accessed as a text stream; when a binary stream is used, a null input line is always the end-of-file indicator, and the amparm is ignored.

  Use of uppercase and lowercase letters is significant. For example, if **`eof=END`** is specified, an input line of **`end`** will not be interpreted as the end of file. The default specification for **`eof`** is **`eof=EOF`**; that is, the input line **`EOF`** will be interpreted as the end of file.
- □ The **`prompt`** amparm enables you to specify a string that is sent to the terminal before data are read from the terminal with this stream.Use of uppercase and lowercase letters is significant. If the prompt string ends in a new-line (**`'\n'`**) character under MVS, the cursor is positioned to the line following the prompt; otherwise, the cursor appears on the same line as the character after the prompt. Under CMS, the presence or absence of a trailing new-line character in the prompt is not significant, because the cursor is always positioned to the first position of the terminal input area.

  With one exception, the default value for the **`prompt`** amparm is **`''prompt=''`**; that is, no input prompting occurs. However, when the library opens the standard input stream (**`stdin`**), a prompt consisting of the program name followed by a colon is used. See “stdin, stdout, and stderr” on page 3-59 for information on overriding this default.

Using the * filename under the shell

As an aid to porting existing programs to the OpenEdition shell, the SAS/C library allows the filename `*` in `ddn`, `dsn`, or `tso` style to access the OpenEdition terminal. When you use this filename, the `eof` and `prompt` amparms are permitted and honored. These amparms are ignored when you open the HFS file `/dev/tty`. Use of these amparms under the shell is not recommended for new programs, because they are not portable, but they can be useful when porting existing MVS programs.

Note: When a program running under the shell opens the `*` filename, no distinction is made between text and binary access. The effect of control characters on the display is defined by OpenEdition.

Using the OpenEdition Hierarchical File System

The OpenEdition Hierarchical File System (HFS) is an implementation of a UNIX file system under MVS. In this file system, a directory is a special kind of file that contains names and other information about a group of files. The *root* directory is at the top of the hierarchy; thus, the root directory is not contained in any other directory. Files within the file system are identified by a pathname, which consists of the series of directories (beginning with the root directory) that lead to a file. Directory names are separated by slashes (/), and the filename itself comes last. For example, the pathname `/u/marie/tools/wrench.c` identifies the file `wrench.c`, contained in the directory `tools`, in turn contained in the directory `marie`, contained in the directory `u`, which is contained in the root (/) directory. This type of pathname, beginning with a slash connoting the root directory, is called an *absolute pathname.*

A program using a hierarchical file system always has a current directory defined, either by inheritance from a calling program, or from using the `chdir` function. A pathname without a leading slash is called a *relative pathname.* Such pathnames are interpreted relative to the current directory. For instance, if a program's current directory is `/u/marie` and it attempts to open the pathname `tools/wrench.c`, the file that is actually accessed is `/u/marie/tools/wrench.c`.

Note: When you call the `fopen` or `open` function to access an HFS file, it may be necessary to prefix the pathname with the SAS/C style prefix `hfs:`. See "File Naming Conventions" on page 3-56 for information on when this is required.

Several different kinds of files exist in the hierarchical file system. Most files are so-called *regular files*, which are stored on disk (in a special MVS file system data set). The hierarchical file system also contains special files of various sorts, which may not have all the properties of regular files. For instance, some special files do not support seeking, or have different behavior when read or written. Some important examples of special files include

- the user's terminal (named `/dev/tty`).
- the null device (`/dev/null`). Output to `/dev/null` is discarded; input from `/dev/null` produces end-of-file.
- pipes, which are files used to communicate between processes. OpenEdition supports both named and unnamed pipes.

Low-level and Standard I/O

I/O to the hierarchical file system is implemented by OpenEdition MVS via a set of services that correspond to traditional UNIX unbuffered I/O calls, such as `open`, `read`, and `write`. For HFS files, UNIX style I/O functions interface directly to the operating system, bypassing most of the C library's I/O support. This ensures that access to the Hierarchical File System through SAS/C has the same characteristics as access when the operating system interfaces are used directly.

When an HFS file is opened using **open**, the operating system returns a small integer representing the file, called the *file descriptor*. All other I/O operations, such as reading and writing, are performed by specifying the file descriptor. File descriptors have two important properties not applicable to more traditional MVS files:

- They may be shared between programs. When several programs are reading or writing the same file, the results are well-defined; whereas, with other MVS file types, the results are undefined, and generally undesirable. See the function description of **fcntl** for information on how several programs sharing a file can cooperate to avoid interfering with each other.
- They can be inherited by a program from its caller. For example, the program **dict1** could open a file using file descriptor 4, and then call the program **dict2** using the **execvp** function without closing file descriptor 4. When **dict2** began execution, file descriptor 4 would still be open, with the same file position as at the time of the **exec**, and **dict2** could immediately read or write the file without having to open it again.

Of course, you can use standard I/O functions rather than the low-level functions like **open**, **read**, and **write** to access HFS files. However, program behavior may differ, depending on which set of routines you use. When you use **fopen** to open an HFS file, it calls the OpenEdition open interface, and then saves the resulting file descriptor in a control block accessed via the **FILE** pointer. Functions such as **fread** and **fwrite** read and write data from a buffer area allocated by the library (or by the user if the **setvbuf** function is used), and actually read from or write to the file descriptor only as necessary to empty or fill the buffer.

For most programs, the buffering performed by standard I/O results in a performance gain, because the program does not need to call the operating system as often. However, for some programs, this can result in unacceptable behavior. For example, programs that share files usually should not use standard I/O because output data may be buffered indefinitely; therefore, updates may not become visible to other programs using the file for an arbitrary amount of time. Similarly, if a program needs to receive an open file from a calling program, it must be aware that only the file descriptor is passed. That is, a **FILE** pointer is local to the program that creates it, and it cannot be inherited, except under special conditions.

For applications that might need to access a file using both low-level and standard I/O, the POSIX standards define two functions that cross the boundaries:

- Use **fdopen** to associate a **FILE** pointer with an open file descriptor. For example, if the program **dict2** receives open file descriptor 4 from its caller, it can use the following statements to associate the **FILE** pointer **f** with file descriptor 4. Thereafter, the program uses standard I/O functions to access the file.

  ```
  FILE *f;
  f = fdopen(4, "r+");
  ```

- Use **fileno** to extract the file descriptor for an HFS file from the **FILE** pointer. This can be useful if a program using standard I/O has a momentary need for a low-level I/O feature not supported via standard I/O, such as the **ftruncate** function.

When a program is called by **exec**, the library automatically uses **fdopen** to associate the standard files **stdin**, **stdout**, and **stderr** with file descriptors 0, 1, and 2. Thus, these three files are partial exceptions to the rule stated earlier that **FILE** pointers cannot be inherited across **exec**.

OpenEdition I/O Considerations

OpenEdition support in SAS/C affects I/O in several ways. SAS/C now implements two different file-naming conventions. Also, DD statements can now be allocated to HFS files or directories. Finally, with OpenEdition support, you may find it useful to modify some programs to use PDS members and HFS directories interchangeably. These considerations are described in the next three sections.

File Naming Conventions

SAS/C implements two different file-naming conventions: one for use by traditional SAS/C programs, and one for POSIX-oriented programs. The choice of naming convention depends on whether any compilation in the main program load module specifies the `posix` compiler option. If so, then POSIX file-naming rules apply. If no compilation specifies the `posix` option, then traditional SAS/C naming conventions apply.

Using traditional SAS/C rules, a filename consists of a style prefix (one to four characters, followed by a colon), followed by the filename proper. The prefix determines how the rest of the filename is to be interpreted (for example, as a DDname or an HFS pathname). If there is no style prefix, then a default prefix is assumed. The default prefix may be defined by the program by initializing the `_style` external variable. If `_style` is not initialized, the default is system-dependent. A filename, with or without an explicit style prefix, may be further prefixed by the string `//`. If `//` precedes a style prefix, the `//` is simply ignored. If `//` is present, but there is no style prefix, then the style `tso` (in MVS) or `cms` (in CMS) is assumed, independent of the `_style` definition. When these rules are in effect, you must do one of the following to access a file in the HFS:

- Prefix the path name with `hfs:` or `//hfs:`. For example, to access the file `tools/wrench.c` in the current directory, open `hfs:tools/wrench.c` or `//hfs:tools/wrench.c`.
- Initialize the `_style` external variable to `''hfs''`. Then simply use the pathname to open HFS files. However, you will have to use a style prefix for other kinds of names, such as DDnames.

The rules above are useful for MVS-oriented programs, or for programs that must open diverse kinds of files. However, they are often not the most appropriate rules for portable applications. Notably, the POSIX.1 standard requires that any pathname not beginning with `//` be interpreted as a hierarchical file system pathname. For this reason, SAS/C implements alternate conventions to allow the porting and/or development of applications that conform to the POSIX.1 standard and are portable to UNIX operating systems.

These alternate rules apply whenever the main load module of a program contains at least one compilation using the `posix` compiler option. For such programs, the file-naming conventions are as follows:

- If the name of the file to be opened does not begin with exactly two slashes (`//`), it is the pathname of an HFS file, even if the name appears to have a style prefix. For example, the filename `/u/marie/tools/wrench.c` identifies an HFS file with that pathname, and `ddn:sysin` identifies a file with that name in the current directory.
- If the name of the file to be opened begins with two slashes (`//`), it is interpreted exactly as it would be interpreted according to the traditional SAS/C rules above. That is, the filename `//ddn:sysin` identifies the DDname SYSIN, and the filename `//tools.c(wrench)` identifies the MVS PDS member `userid.TOOLS.C(WRENCH)`, where `userid` is the current user's id.

Note: For a program compiled with the `posix` option, the `_style` external variable is ignored.

Note: Because filenames beginning with `//` are interpreted in the same way for applications compiled with the `posix` option as for those compiled without the `posix` option, this form should be used by any functions that need to open files, and which can be used in programs compiled with or without the `posix` option. For example, to open the HFS file `/u/marie/tools/wrench.c` without knowing whether the program was compiled with the `posix` option, use the filename `//hfs:/u/marie/tools.wrench.c`. Any such functions must not be compiled with the `posix` option themselves, because then any program using such functions would automatically follow the naming conventions for programs compiled with the `posix` option.

Accessing HFS files using DDnames

Enhancements to MVS JCL and dynamic allocation facilities for OpenEdition MVS allow DD statements to be allocated to HFS files or directories. Parameters on the DD statement correspond roughly to arguments to the `open` function: the PATH option corresponds to the pathname to be opened, the PATHOPTS option corresponds to the `open` flags, and the PATHMODE option corresponds to the file creation mode specification.

HFS files can be accessed using a `ddn`-style filename, as can any other MVS file. The following points should be noted:

- The `open` or `fopen` options must be compatible with the DD statement. For example, if the DD statement specifies PATHOPTS=ORDONLY, but the `fopen` call specifies a mode of `'r+'`, the open will fail.
- If you specify PATHOPTS=OCREAT on the DD statement or allocation and the specified file does not exist, the file is created at the time of allocation. This means that at the time the program calls `open` or `fopen`, the file already will have been created. In particular, if the DD statement specifies PATHOPTS=(OCREAT,OEXCL) and the `open` call also specifies `O_CREAT+O_EXCL`, the `open` will fail, because the file will have been created when the DD statement was processed.
- DD statements that reference directories cannot be opened.
- Concatenated DD statements in which one or more members are HFS files cannot be opened successfully.

Using HFS directories and PDS members interchangeably

Until the availability of OpenEdition, it was often convenient to replace the use of directories in UNIX applications with PDS's when porting them to MVS. Consider porting a UNIX C compiler to the mainframe. In UNIX, a system header file like `<stdio.h>` is simply a file in a particular directory. In MVS, such names are generally interpreted by treating the first part of the name as a member name, relative to a PDS defined by a DDname. (For example, SAS/C interprets `<stdio.h>` as `ddn:syslib(stdio)`). With the availability of OpenEdition, it may be desirable to modify these programs to use a PDS or an HFS directory interchangeably, as convenient for the user. SAS/C provides the following extension to its `ddn`-style filename handling in support of this. Besides all previously accepted forms, a `ddn`-style filename may now have the following form:

```
ddname/filename
```

Here, **ddname** is a valid MVS filename, and **filename** is a valid POSIX filename (not containing any slashes). When **ddn:ddname/filename** is opened, the following occurs:

- □ If the **ddname** is defined and allocated to an HFS directory **dirname**, the file **dirname/filename** is opened. For example, if the DDname SYSLIB references the directory **/usr/include**, then opening **ddn:syslib/stdio.h** is the same as opening **hfs:/usr/include/stdio.h**.
- □ If the **ddname** is defined and allocated to an MVS PDS, the file **ddname(member)** is opened, where the member name is the same as **filename**, discarding the first period in the name and all succeeding characters, and truncating the remainder of the name to eight characters. For example, if the DDname SYSLIB references a PDS, then opening **ddn:syslib/stdio.h** is the same as opening **ddn:syslib(stdio)**.
- □ If the **ddname** is undefined, or it references some other kind of file, the open fails.

Note: When the **ddname/filename** syntax is used and the DDname references an HFS directory, any PATHOPTS specified on the DD statement apply to the subfile as well. Thus, if DDname SYSLIB specifies PATHOPTS=OWRONLY, opening **ddn:syslib/stdio.h** using open mode **'r'** will fail.

Using Environment Variables in place of DDnames

When a new process is created by **fork** or **exec**, as when a program is called by the shell, a new address space is created with no DD statements allocated other than possibly a STEPLIB. For programs exclusively using UNIX oriented interfaces, this does not present a problem, but it can present difficulties for porting existing MVS applications to run under the shell. For this reason, the SAS/C library permits you to substitute environment variables for DDnames in programs invoked by the **exec** system call.

For a program invoked by **exec**, if an attempt is made to open a DDname (for example, using the filename **//ddn:anyfile**), if no corresponding DD statement exists, the library checks for an environment variable named **ddn_ANYFILE**. Notice that the prefix **ddn** is always in lowercase letters, while the DDname proper is always in uppercase letters. The value of the environment variable, if it exists, must have one of two forms:

- □ If the environment variable value does not begin with a slash, the value is translated to uppercase letters and then interpreted as a fully qualified MVS dataset name. For example, if the value of **ddn_MACLIB** is **sys1.maclib(dcb)**, an **fopen** of **//ddn:maclib** is treated as if the call specified **//dsn:sys1.maclib(dcb)**. However, any MVS dataset specified via a **ddn_** environment variable must already exist; that is, the library will not create a new data set while processing an environment variable. However, you can reference a nonexistent member of an existing PDS.
- □ If the environment variable begins with a slash, the value is interpreted as an HFS absolute pathname. For example, if the value of **ddn_MACLIB** is **/usr/include/stdio.h**, an **fopen** of **//ddn:maclib** is treated as if the call specified **//hfs:/usr/include/stdio.h**. A **ddn_** environment variable can reference a nonexistent HFS file, which will then be created when the DDname is opened (if permitted by the **fopen** options).

When a **ddn**-style filename is opened using an environment variable, the specified DDname is allocated by the library during processing. Thus, if the same program opens the DDname a second time, a DD statement will be found, and the environment variable will not be referenced again. Consequently, changing the environment variable after it has been used to open a file will be ineffective.

Note: The `ddn:ddname/filename` pathname format described above can be used both with DDnames defined by an environment variable and with actual DD statements.

File descriptor allocation

Whenever a file is opened using the **open** system call, the POSIX.1 standard requires that the call be assigned the lowest file descriptor number that is not in use by an open file. Under OpenEdition, the range of valid file descriptors is from 0 to a maximum defined by the site. The default maximum is 64, but it can be set by the site to be as low as 16 or as high as 65,536. The maximum number of open OpenEdition files can be determined using the **sysconf** function.

The limit on the number of open file descriptors is unrelated to the library's limit on the number of **FILE** pointers that may be opened using standard I/O. This limit is always 256, regardless of the OpenEdition limit. File descriptors in the valid OpenEdition range can be assigned to files other than OpenEdition files in two situations:

- The library treats the **FILE** pointers **stdin**, **stdout**, and **stderr** as being file descriptors 0, 1, and 2, whether or not these are HFS files.
- The SAS/C socket library assigns sockets file descriptor numbers in the OpenEdition range, because many socket programs assume that socket numbers are allocated using the rules for UNIX file descriptors.

In both of these cases, confusion can occur. For example, if file descriptor 4 is assigned to a socket and you call **open**, OpenEdition could assign file descriptor 4 to the newly opened file, and then the library could not distinguish a request to write to file 4 from a request for socket 4.

The library solves this problem using shadow files. Whenever the library needs to assign a file descriptor for a file that is not an OpenEdition file, it first opens **/dev/null** to obtain a file descriptor, which is then assigned to the socket. The shadow file is closed only when the socket or standard file is closed. Because OpenEdition associates the file descriptor with **/dev/null**, it will not be possible for OpenEdition to associate the descriptor with any other file. This technique also ensures that socket numbers are assigned in accordance with OpenEdition rules.

You should note the following points about file descriptor allocation:

- This technique means that it is not possible to use more sockets and OpenEdition files combined than the maximum number of OpenEdition file descriptors. If this is a problem, it should be solved by raising the site file descriptor limit.
- When you use the **open** function to open MVS files for UNIX style I/O, very large file descriptors are assigned, thereby preventing these files from affecting the OpenEdition file limit.
- If you run more than one program that uses OpenEdition facilities in the same address space, they share all open file descriptors, except when a new process is created using **oeattach**. In such cases, file descriptors may not be assigned in the order specified by POSIX.1. This mode of operation is not recommended, because the sharing of file descriptors (and other data, such as signal handlers) between the two programs can lead to very confusing results.

stdin, stdout, and stderr

The C language definition specifies that when program execution begins, three standard streams should be open and available for program use. These are **stdin**, the standard input stream, **stdout**, the standard output stream, and **stderr**, the standard error stream. A number of C library functions, such as **puts** and **scanf**, are defined to use **stdin** or **stdout** automatically, without requiring you to explicitly specify a **FILE** pointer. Note that the standard streams are always opened for text access.

stdin, **stdout**, and **stderr** are implemented as macros, not as true variables. For this reason, you cannot assign them new values. If you want to reopen one of the standard streams, you must use the **freopen** or **afreopen** function rather than **fopen** or **afopen**.

Whether the standard streams are actually used is determined by the program, with one exception. Library diagnostic messages are written to **stderr**, if it can be opened successfully and is suitable for output. If **stderr** is unavailable, library diagnostics are written to the terminal under CMS or TSO, and to the job log under MVS-batch.

Under CMS, all three standard streams are, by default, directed to the terminal. Under MVS, the default filenames for **stdin**, **stdout**, and **stderr** are **ddn:sysin***, **ddn:sysprint***, and *****, respectively. **stdin** uses the DDname SYSIN, if it is defined, and the terminal, otherwise. Similarly, **stdout** uses SYSPRINT, if it is defined, and the terminal, otherwise. **stderr** is directed to the terminal or to the DDname SYSTERM, if running in batch.

For a program running under OpenEdition MVS, by default **stdin**, **stdout**, and **stderr** are defined as file descriptors 0, 1, and 2, as passed by the calling program. If one or more of these file descriptors is not open in the calling program, any attempt to use the corresponding standard file in the called program will fail, unless it opens the appropriate file descriptors itself.

Under MVS, it is possible for one or more of the standard streams to fail to open. For instance, in batch, **stdin** cannot be opened unless you define the DDname SYSIN, and **stderr** cannot be opened unless you define the DDname SYSTERM. To avoid generating an "open failure" error message for a file that is never used, the library delays issuing a system open for a standard stream until it is first used. Note that opening a file under MVS requires significant memory. For this reason, if you write to a standard file when your program runs out of memory (for instance, when **malloc** fails), you may want to force the file to be opened earlier, as by writing an initial new line at a time when enough memory is known to be available.

Changing standard filenames at execution time

Because the standard streams are initialized by the library before execution rather than by an explicit call to **fopen**, there is no direct way to change the filenames associated with them. For this reason, C implementations traditionally support command-line redirection. This permits the user of a program to specify on the command line (that invokes the program) the filenames to be associated with standard input and output streams. For example, the CMS command line **''xyz <ddn:input >printer''** invokes the program XYZ, requesting that **ddn:input** be used as the filename for **stdin**, and that **printer** be used as the filename for **stdout**. Redirection is described in detail in *SAS/C Compiler and Library User's Guide.* Additionally, you should be aware of the following considerations:

- Even when redirection is used, the standard streams are not opened by the operating system until necessary. Therefore, any errors in the filename specified by the redirection are not detected until the file is used. If the operating system cannot open the file, the program treats it like any other I/O error. You should call the **ferror** function to test for errors using a standard stream, just as for any other stream, to avoid wasting time trying to read or write a file that cannot be accessed.
- Names specified with redirection that do not include a specific style prefix are ordinarily assumed to be DDnames under MVS, or **cms** style filenames under CMS. You can initialize the **_style** external variable to define a different default style, as described in Chapter 9 of *SAS/C Compiler and Library User's Guide.* The default style applies to all files used by the program, not just to the standard files.

- When a program is invoked by the OpenEdition shell, redirections are handled by the shell, not by the SAS/C library. This means that redirections must be in the format defined by the shell, not by the SAS/C library. In particular, you cannot use a style prefix in a redirection for a program invoked by the shell.

Changing standard filenames and characteristics at compile time

Besides supporting command-line redirections, the library enables you to change the names of the standard files at compile time, or to specify amparms to be used when the files are opened. Thus, you can override some of the library defaults. If the program specifies a replacement filename and the command line includes a redirection for the same file, the filename specified on the command line is used.

To change the default name for a standard file, you must initialize an external **`char *`** variable with the filename to be used. The external variables are **`_stdinm`**, **`_stdonm`**, and **`_stdenm`** for **`stdin`**, **`stdout`**, and **`stderr`**, respectively. For example, the following declaration specifies that by default, **`_stdinm`** should read from the user's virtual reader:

```
char *_stdinm = "cms:reader";
```

The **`_stdinm`**, **`_stdonm`**, and **`_stdenm`** specifications are honored even for programs called with **`exec`**. Thus, using these variables, you can override the standard use of file descriptors 0, 1, and 2 for these files if you wish. If you do this, the standard file descriptors are not closed, and can still be accessed directly via the file descriptor number.

Similarly, you can assign an initial value to the external variables **`_stdiamp`**, **`_stdoamp`**, or **`_stdeamp`** to specify the amparms to be used when **`stdin`**, **`stdout`**, or **`stderr`** is opened. The library default amparms are shown in Table 3.7:

Table 3.7
Default Amparms for the Standard Files

File	**Amparms**
`stdin`	`prompt=`*`pgmname`*`:\n`
`stdout`	`print=yes`
`stderr`	`print=yes,page=60`

You may want to override these default amparms in the following situations:

- If **`stdin`** is defined as the terminal, a prompt of the form ***`pgmname`***: (where ***`pgmname`*** is the program name or **`""`** if the program name cannot be determined) is issued to the terminal before each read. If your program performs its own prompting, you may want to initialize **`_stdiamp`** to **`""`** to suppress the library prompt.

  Note: A standard prompt is not used when **`stdin`** is defined as file descriptor 0 (for a program called by **`exec`**), even if file descriptor 0 references the terminal.
- Because the default **`stdout`** and **`stderr`** amparms include **`"print=yes"`**, the library issues a warning message if the associated physical file does not support page formatting (for example, if it is an MVS data set whose record format does not include A). If you expect your program to be run with **`stdout`** or **`stderr`** associated with this type of file, you can initialize **`_stdoamp`** or **`_stdeamp`** to **`"print=no"`** to inhibit the diagnostic message.

Using the standard streams with UNIX style I/O

In UNIX operating systems and other similar systems, it is possible to access the standard streams using low-level I/O, specifying file numbers 0, 1, and 2 for **`stdin`**, **`stdout`**, and **`stderr`**, respectively. The library supports such access, provided that certain guidelines are followed. This usage is nonportable. The following restrictions apply:

- Use file number 0 (**`stdin`**) for input only, and file numbers 1 (**`stdout`**) and 2 (**`stderr`**) for output only.
- Do not use **`lseek`** on any of these files.
- Do not close any of these files.
- Avoid using the same file with both UNIX style I/O and standard I/O. For instance, do not issue both **`read`** to file 0 and **`fgetc`** to **`stdin`** in the same program.
- When OpenEdition is in use, it is possible to create confusing associations between file descriptors in certain circumstances. For example, it is possible to cause file descriptor 0 to be associated with **`stdout`**, rather than with **`stdin`**. If you call a UNIX I/O function with a standard file descriptor that is not assigned by OpenEdition, and whose corresponding standard **`FILE`** pointer is associated with a different file descriptor, the library will reject the call rather than possibly access the wrong file.

I/O Error and Interrupt Handling

UNIX style I/O includes no specific error-handling functions or features. If a **`read`**, **`write`**, or **`lseek`** call fails, the only indication is the value returned by the function. Depending on the error, it may be possible to continue to use the file after the error occurs.

Error handling

As stated earlier, after a file has been opened, a pointer to a **`FILE`** object is used to identify the file. This pointer is passed to I/O routines such as **`fread`** and **`fwrite`** to indicate the file to be read or written. Associated with each **`FILE`** object is a flag called the *error flag* that indicates whether the most recent I/O request failed. When the error flag is set, it is not possible to use the file other than to close it or to call the **`clearerr`** function to clear the flag.

The error flag for a file is set whenever an error occurs trying to access a file. The flag is set for all types of errors, whether they are hardware errors (such as an unreadable tape block), errors detected by the operating system (such as a full CMS minidisk), or errors detected by the library (such as trying to read a write-only file). In addition to setting the error flag, the library also writes a diagnostic message to the **`stderr`** stream and sets the **`errno`** external variable to indicate the type of error that occurred.

The function **`ferror`** can be called to determine whether the error flag is set for a file. Using this function is sometimes necessary because some functions, such as **`fread`**, do not distinguish in their return values between error conditions and end of file.

If you want to continue processing a file after an error occurs, you must call the **`clearerr`** function to clear the error flag; that is, to cancel the effect of the previous error. Some errors (such as running out of disk space under MVS) are so severe that it is impossible to continue to use the file afterwards without reopening it. In such cases, **`clearerr`** is unable to clear the error, and continued attempts to use the file cause new errors to be generated.

I/O and signal processing

In a program that handles asynchronous signals, it is possible for a library I/O routine to be interrupted by a signal. When a library I/O routine is interrupted, an interrupt flag is set for the file until the signal handler returns. Any attempt to use the file while the interrupt flag is set is treated as an error (and therefore sets the error flag) to avoid damage to the file or to library file control blocks. The situations in which the interrupt flag is most likely to be set are after using **longjmp** to exit from a signal handler, or when a signal handler performs I/O to a file in use at the time of the signal. When the interrupt flag is set, you can call **clearerr** to clear it along with the error flag and continue to use the file.

For terminal input under MVS and CMS (except with OpenEdition), the system calls do not allow signals to be detected while the program is waiting for terminal input, with one exception. The SIGINT signal, which is an attention interrupt under MVS or an IC immediate command under CMS, terminates the terminal read and causes any handler to be called immediately. If your SIGINT handler needs to read from the terminal, you should use a different **FILE** pointer from the one used by the rest of your program; otherwise, the error flag is set for the file, as described in the previous paragraph. If you must use the same **FILE** pointer in mainline code and in your handler, you need to call **clearerr** in the handler before reading from the terminal and call it again after exit (either by **return** or by **longjmp**) from the handler.

Augmented Standard I/O

Some 370 I/O applications are beyond the scope of standard I/O because the record concept is absent from the C language. Consider, for example, a program to make an exact copy of any input file, including duplicating the input file's record structure. Such a program could not be written using binary file access because all information about the record structure of the input file would be lost. It also could not be written using text access, because if there were any new-line characters in the input file, they would be interpreted by the program as record breaks, and the output file would contain more records than the input file. The functions **afread**, **afread0**, **afreadh**, **afwrite**, **afwrite0**, and **afwriteh** have been defined to permit this sort of application to be written in C. These functions, together with **afopen**, **afreopen**, and **afflush** are known as *augmented standard I/O.*

afread and **afwrite** can only be used with binary streams. Because they are used with binary streams, they never translate or otherwise modify input or output data, even if the data include control characters. **afread** and **afwrite** are useful only when the **''seq''** access method is used, because a file processed with the **''rel''** access method is treated as a stream of characters without record boundaries. If you need to process files with fixed-length records using **afread** or **afwrite**, you should open the file with **afopen**, and request the use of the **''seq''** access method.

afread and afwrite

The **afread** and **afwrite** functions are very similar in form to the standard **fread** and **fwrite** functions: they accept as arguments a pointer to the input or output area, the size of the type of object to be read or written, the maximum number of objects, and the **FILE** pointer identifying the file. But, unlike **fread** and **fwrite**, whose purpose is simply to read or write the items specified without regard to record boundaries, the purpose of **afread** and **afwrite** is to read or write the items specified as a single record. Specifically, **afread** and **afwrite** read and write items as follows:

- When **afread** is called, it reads items from the file until a record boundary is encountered. It reads, at most, the number of items specified, and it generates a diagnostic message if there are any further items in the record. It is permitted for the input record to contain fewer items than requested. In this case, **afread** reads as many as are present in the record, and returns the number of items read to its caller.

This permits easy processing of files containing variable-length records with **afread**.

- ☐ When **afwrite** is called, it writes all the items specified and then forces a record break to occur. An error message is generated if the items do not all fit in a single record, or if the file characteristics will not permit writing a record of that size.

afread and **afwrite** do not support zero-length records. On input, a zero-length record is ignored, and similarly, an attempt to write a zero-length record is ignored. Two alternate functions, **afread0** and **afwrite0**, are provided. These functions can handle zero-length records, if the file being processed supports them. To support zero-length records, **afread0** and **afwrite0** use error-reporting conventions that are not compatible with the standard C **fread** and **fwrite** functions.

afread and **afwrite** do not require that the file be positioned to a record boundary when they are called. Also, you can freely mix calls to **afread** and **afwrite** with calls to other standard I/O routines, such as **fscanf** or **fseek**, if your application requires it. See the function descriptions for **afread** and **afwrite** for examples of their use.

afreadh and afwriteh

afreadh and **afwriteh** enable you to read or write a header portion of a record before calling **afread** or **afwrite** to process the remainder. This is useful for reading or writing files processed by another language (such as COBOL or Pascal) that supports variant records.

A *variant record* is a record composed of two parts, a fixed format part and a variable format part. The fixed format part contains information common to all records, and a field defining the length or structure of the remainder of the record. Depending on the situation, it may not be possible to read or write such records conveniently using **afread** and **afwrite**. (Defining the records to be processed as a C union is helpful only if all the different variants are the same size.) **afreadh** and **afwriteh** support processing such records in a straightforward way:

- ☐ **afreadh** is similar to **afread**, except it does not require that a record break occur after the last item read. However, all items read must be contained in a single record, or an error message is generated. **afreadh** is most frequently used to read the first part of a variant record.
- ☐ **afwriteh** is similar to **afwrite**, except it does not force a record break after the last item written. However, all the items written must fit into a single record, or an error message is generated. **afwriteh** is most frequently used to write the first part of a variant record.

See the function descriptions for **afreadh** and **afwriteh** for examples of their use.

Advanced MVS I/O Facilities

This section discusses several advanced I/O tasks under MVS, such as reading a PDS directory, recovering from ABENDs, PDSE access, and processing DIV objects.

Reading a partitioned data set directory

You can read a PDS directory sequentially by allocating the entire library to a DDname, and specifying the DDname without a member name, as the filename. For instance, you can use the following TSO code fragment to open the directory of SYS1.MACLIB for input:

```
system("tso:alloc file(sysmacs) da('sys1.maclib') shr");
direct = fopen("ddn:sysmacs", "rb");
```

You can also access the PDS directory by opening the PDS using a **``dsn''**- or **``tso''**-style name, and specifying the amparm **``org=ps''**, as in

```
   direct = afopen("dsn:sys1.maclib", "rb", "seq", "org=ps");
```

The directory is treated by the library as a RECFM=F, LRECL=256 data set, regardless of the attributes of the members.

You cannot use C I/O to modify a PDS directory. Also, access to a PDS directory is supported using only **ddn**-style filenames, unless the **org** amparm is used. If you specify a PDS using a **dsn**- or **tso**-style filename without an **org** specification, and no member name is present, the member name TEMPNAME will be used.

Recovering from B37, D37, and E37 ABENDs

When an I/O operation requires additional space to be allocated to a file but space is unavailable, the program is normally terminated by the operating system with a B37, D37, or E37 ABEND. The SAS/C library intercepts these ABENDs and treats them as error conditions. It sets the error flag for the affected file and returns an error code from the failing I/O function. The ABEND is intercepted using a DCB ABEND exit, not a STAE or ESTAE, and functions correctly even if you use the **nohtsig** run-time option to suppress the library's ESTAE.

When the library recovers from one of these ABENDs, the file is automatically closed by the operating system. For this reason, the error flag is set permanently; that is, you cannot clear the flag with **clearerr** and continue to use the file. An exception is made by the **``rel''** access method, which reopens the file if you use **clearerr** to clear the error condition. This enables you to read or modify data you have already written, but you cannot add any more records to the file, because this simply will cause the ABEND to reoccur.

Although other kinds of I/O errors are quite rare, these out-of-space ABENDs occur frequently, even for production programs. Therefore, you should always check output operations for success to avoid loops when trying to write to a file that can no longer be accessed.

Using a PDSE

Recent releases of MVS/ESA have introduced a new implementation of extended partitioned data sets, called a PDSE (Partitioned Data Set Extended). These files are compatible with ordinary PDS data sets, but have a number of advantages, including the following:

- □ Space within a PDSE is allocated dynamically, so PDSEs do not require compression.
- □ Several members of a PDSE can be written at the same time. Different programs can write different members of the same PDSE without interfering with each other.
- □ The directory for a PDSE does not have to be allocated in advance; therefore, a PDSE can expand indefinitely without running out of directory blocks.

The SAS/C Library includes support for PDSEs. Most programs that presently access PDS members can access PDSE members without change.

Restrictions

Although PDSEs are compatible in most ways with standard PDSs, they do not support either BSAM INOUT or OUTIN processing, which enable a member to be read and written at the same time. When the `fseek` or `fsetpos` functions are used on a PDS member, the library depends on this processing, except for a read-only file. For this reason, the use of `fseek` or `fsetpos` on a PDSE member is not supported unless the member is read-only, or unless you specify `grow=no`. One exception is that `fseek(f, 0L, SEEK_SET)` can always be used to reposition a PDSE member to the start of file.

Note: When a PDSE member is accessed through UNIX style I/O in binary mode, this restriction does not apply. In this case, full use of the `lseek` function for repositioning is supported.

Access via the grow= amparm

The SAS/C Library defines the amparm `grow`, which can be specified when a file is opened for `'r+'` or `'r+b'`. You specify `grow=no` to inform the library that the program will only replace existing records of a file, rather than adding any data to the end. When you specify `grow=no` for a PDSE member, the library can open the member for UPDAT rather than OUTIN and can then support use of either the `fseek` or `fsetpos` function.

The `grow` amparm is also supported for standard PDS members, and it should be used where possible, because it performs an update-in-place action, and avoids wasting the space in the PDS occupied by the previous member.

Allocating PDSEs

When a new partitioned data set is created, the decision to create it as a regular PDS or as a PDSE is normally determined by your site, possibly based on data set name or other data set characteristics. In some cases, you may want to force a particular choice. The `org` amparm supports this. `org` has more uses than just PDS allocation. See "Opening Files" on page 3-27 for more information.

When you use the `afopen` function to create a new PDS, you can specify one of three values for `org`

`po`	specifies that the file is a PDS and that normal site criteria should be used to select between a regular PDS and a PDSE.
`pds`	specifies that the file should be created as a regular PDS.
`pdse`	specifies that the file should be created as a PDSE.

Note: A site may choose to ignore a program's request for a particular type of PDS, although this is fairly unusual. For this reason, it cannot be guaranteed that `org=pds` or `org=pdse` will be honored in all cases. If your operating-system level does not support PDSEs, the `org` values `pds` and `pdse` will be treated like the value `po`.

Using VSAM linear data sets (DIV objects)

A DIV object is different from other MVS-type data sets. Essentially, it is a single stream of data with no record or block boundaries. The operating system processes the file in 4096-byte units with paging I/O, mapping the data in the file to virtual storage referred to by the program (all of which is transparent to the program). For more information on DIV objects, see the IBM manual *MVS/ESA Application Development Guide.*

You can access DIV objects using the ordinary C library I/O functions and the ``rel'' access method. Two amparms are available for use with VSAM linear data sets. These amparms are not required, but they allow the program to direct the internal buffering algorithm used by the library:

`bufsize=`*nnn*	specifies the size, in bytes, of a DIV window.
`bufmax=`*n*	specifies the number of DIV windows.

The value specified for **bufsize** is rounded up to a multiple of 4096. The default value for **bufsize** is **bufsize=262144** (256K). The default value for **bufmax** is **bufmax=4**. These default values can be modified by your site; see your SAS software representative for C compiler products for more information about the default values for **bufsize** and **bufmax**. This discussion assumes the default values have not been modified.

DIV windows The library allocates one window when the object is opened. This window is mapped to the beginning of the object. When a reference is made to a location that is outside the bounds of the window, the library allocates a new window that maps the location.

New windows can be allocated, until the number specified by **bufmax** is reached. Then, if a reference is made to a location that is not mapped by any window, the library remaps the least-used window to the new location. The least-used window is the window that has the fewest references made to locations that it maps.

If the limit specified by **bufmax** has not been reached, but there is insufficient storage available to allocate a new window, the library issues a warning and begins remapping existing windows.

How the amparms are used As with other amparms, the linear data set amparms may be specified in the **amparms** argument to **afopen** or **aopen**. If one of the amparms is omitted, then the library uses its default value.

If a linear data set is opened with **fopen** or **open**, or neither amparm is used, then **bufsize** is calculated from the object size divided by 4, rounded to a multiple of 4096 as necessary. If the data set has size 0 (that is, the data set is empty), the default values are used. If there is insufficient storage available to allocate the first window, the library issues a warning and uses whatever storage is available.

Advanced CMS I/O Facilities

This section discusses several advanced I/O tasks under CMS, such as use of **xed** style files, extending global MACLIB/TXTLIB processing, and using the CMS shared file system.

The xed filename style

The CMS version of the library supports access to files being processed by XEDIT with the **xed** filename style. **xed** style filenames have the same format as **cms**-style filenames, for example, **xed:payroll data a**. The filename must identify a CMS disk file. That is, you cannot specify device names such as PRINTER. Also, you cannot use the **MEMBER** keyword (or its abbreviated format equivalent).

You can use the **xed** style even when XEDIT is not active. In this case, or when the file requested is not in the XEDIT ring, the file is read from disk.

See the **system** function description and Chapter 2, “CMS Low-Level I/O Functions,” in *SAS/C Library Reference, Volume 2* for information on other facilities that may be useful for programs that use XEDIT files.

Extensions to global MACLIB/TXTLIB processing

As described previously, you can use the **`cms`**-style filenames **`%MACLIB (MEMBER`** ***`name`***) and **`%TXTLIB (MEMBER`** ***`name`***) to access members of global MACLIBs or TXTLIBs. Global MACLIBs and TXTLIBs are established using the CMS GLOBAL command. Here is an example:

```
GLOBAL TXTLIB LC370 MYLIB1 MYLIB2
```

When **`%TXTLIB(`*`name`*`)`** is opened, the libraries LC370 TXTLIB, MYLIB1 TXTLIB, and MYLIB2 TXTLIB are searched, in that order, for the member ***`name`***. Also, the library implements several extensions to standard CMS GLOBAL processing to support larger numbers of global libraries than allowed directly by the CMS GLOBAL command. These extensions also support the use of OS partitioned data sets as global libraries.

One extension to GLOBAL processing enables you to issue a FILEDEF using the DDname CMSLIB and then include CMSLIB in the list of files for the GLOBAL command. This causes the files associated with the CMSLIB DDname to be treated as global. For example, if you issue the following commands, the same set of libraries as in the previous example is defined, and the effects of opening **`%TXTLIB(`*`name`*`)`** are the same:

```
FILEDEF CMSLIB DISK MYLIB1 TXTLIB A
FILEDEF CMSLIB DISK MYLIB2 TXTLIB A (CONCAT
GLOBAL TXTLIB LC370 CMSLIB
```

One advantage of using the FILEDEF approach is that the FILEDEF may be concatenated, enabling you to bypass the limit of eight global libraries imposed by early versions of CMS. Another is that you can put OS partitioned data sets into the global list (in a non-XA system), as described in the following section. Note that when CMSLIB is concatenated, the global search order is the same as if CMSLIB were replaced in the global list by the files that compose it, in the order in which they were concatenated.

The special processing of the CMSLIB DDname is a feature of the SAS/C library; the DDname CMSLIB has no special significance to CMS.

Using an OS PDS as a global MACLIB/TXTLIB If your site permits CMS access to OS disks, the CMSLIB FILEDEF for use in a global list may reference an OS partitioned data set. The FILEDEF must have the following form:

```
FILEDEF CMSLIB DISK filename MACLIB fm DSN OS-data-set-name TXTLIB
```

The filemode (*fm*) cannot be an asterisk (*) and must refer to an OS disk. The PDS referenced must have fixed-length blocked records with an LRECL of 80. The PDS must reside on a 3330, 3350, or 3380 disk device.

Note: You cannot use an OS PDS as a global MACLIB/TXTLIB in a XA-mode virtual machine.

Using the CMS Shared File System

VM/SP Release 6 introduced a new file system into CMS called the Shared File System (SFS). This file system provides new file management and sharing capabilities. SFS files are stored in a file pool (a collection of CMS minidisks) where users are given space to organize files into directories. Directories enable users to group related files together. By granting read or write authority to files or directories, users can allow other users to share their files. This feature enables several users to have access to the same file at the same time, although only one user can update a shared file at a time.

When a shared file is open for update, the file system provides update access to a copy of the file. Changes to the file do not take effect until the changes are committed. Alternately, after updating a file, the user can roll back the changes, which leaves the file unmodified. If a user opens a shared file for reading while another user is updating it, the reading user accesses a temporary copy of the data and can read only the data in the file at the time it was opened, even after the writing user commits changes.

Shared files can be accessed as if they were normal CMS disk files by using the CMS ACCESS command, which can assign a filemode letter to an SFS directory. Currently, use of unique SFS functionality, such as access to subdirectories and the ability to roll back changes, is not available with the CMS ACCESS command. These features are only available when the Shared File System is used directly.

The SAS/C Library allows access to the Shared File System directly and via the ACCESS command. If you use the ACCESS command to assign a file-mode letter to an SFS directory, files in the directory can be accessed using standard CMS pathnames. Alternately, a shared file can be processed directly by using an **sf**-style filename. For example, opening the following file accesses the file SUMMARY DATA in the directory of userid ACCTING named YR90.JUNE:

```
sf:summary data accting.yr90.june
```

When a shared file is processed directly, it can be committed automatically as it is closed, or the file can be committed explicitly using the **afflush** function.

You can also process an SFS directory as if it were a file (for input only) by using an **sfd**-style filename. This enables you to retrieve various sorts of information about the files or subdirectories stored in the directory. The way in which information is returned is controlled by the **dirsearch** amparm.

SFS files can be processed with either the **''seq''** or **''rel''** access method, if the file attributes are appropriate. Except for **trunc=yes** (which is not allowed), all amparms that can be used with **cms**-style files can be used with an **sf**-style file. SFS directories can be opened only for input, and are always processed by the **''seq''** access method.

For more general information about using the Shared File System, see the IBM publication *VM/ESA CMS User's Guide* and other CMS documentation.

Naming shared files

The format of the name of a shared file, as specified to **fopen**, is

sf:*fileid dirid* [*filemode-number*]

(You can omit the **sf:** prefix if the external variable **_style** has been set to define **sf** as the default style.)

Here *fileid* represents either a standard filename and filetype, or a namedef, which is an indirect reference to a filename and filetype created by the CMS CREATE NAMEDEF command.

Similarly, a dirid represents the following:

[*filepool*]:[*userid*].[*subdir1*.[*subdir2*]...]
namedef]

The *filepool* argument identifies a file pool; *userid* identifies a user; *subdir1*, *subdir2*, and so on name subdirectories of the user's top directory; and *namedef* is an indirect reference to a directory created by the CMS CREATE NAMEDEF command. Note that every dirid that is not a *namedef* contains at least one period. The simplest dirid is `''.''` (which represents the current user's top directory in the default file pool).

Here are some examples of `sf` filenames and their interpretation:

`sf: profile exec .`
specifies the file PROFILE EXEC in the current user's top directory.

`sf: updates amy.`
specifies the file identified by the namedef `updates` in the top directory of user AMY.

`sf: test data qadir 3`
specifies the file named TEST DATA in the directory identified by the namedef `qadir`. The file has file mode 3; that is, it will be erased after it is read.

`sf: graphix data altpool:.model.test`
specifies the file named GRAPHIX DATA in the user's subdirectory MODEL.TEST in the file pool named ALTPOOL.

Note: There is no compressed (blankless) form for `sf` filenames.

Committing changes

When you open an `sf`-style file for update, you control when changes are committed. Two methods are provided to control when changes are committed: the `afflush` function and the `commit` amparm.

The `afflush` function is called to flush output buffers to disk with high reliability. For SFS files, a call to `afflush` causes a commit to take place, so that all changes to the file up to that point are permanently saved.

The `commit` amparm is used with `sf`-style files to specify whether changes will be committed when the file is closed. The default, `commit=yes`, specifies that when the file is closed, changes are committed. The alternative, `commit=no`, specifies that changes are not committed when the file is closed. When you open a file with `commit=no`, you must call `afflush` before closing the file if you want changes saved. On the other hand, if you want to roll back your changes, close the file without calling `afflush`, and no changes will be saved. You can call `afflush` as often as you want, with either `commit=yes` or `commit=no`; when you close a `commit=no` file, all changes since the last call to `afflush` are rolled back. See the `afflush` function description for an example of the use of `commit=no`.

Reading shared-file directories

To process a CMS Shared File System directory, you open an `sfd`-style pathname for input. The pathname specifies the directory to be read, and possibly a subset of the directory entries to be read. The format of the information read from the file, as well as which entries (files and subdirectories) are processed, is determined by the value of the `dirsearch` amparm when the file is opened.

The following values are accepted for `dirsearch`:

`file` specifies that information is to be read for files in the directory. This option corresponds to the FILE option of the CMS DMSOPDIR routine.

`all`	specifies that information is to be read for files in the directory or its subdirectories. This option corresponds to the SEARCHALL option of the CMS DMSOPDIR routine.
`allauth`	specifies that information is to be read for files in the directory or its subdirectories to which the user is authorized. This option corresponds to the SEARCHAUTH option of the CMS DMSOPDIR routine.
`subdir`	specifies that information is to be read for subdirectories of the directory. This option corresponds to the DIR option of the CMS DMSOPDIR routine.

If you specify no value for **`dirsearch`** when you open a shared-file directory, **`dirsearch=allauth`** is assumed.

When you open a directory with **`dirsearch=file`**, **`dirsearch=all`**, or **`dirsearch=allauth`**, the pathname specifies both the directory that is to be read and a filename and filetype, possibly including wild-card characters, indicating from which directory entries are to be read.

The form of an **`sfd`**-style filename for these **`dirsearch`** values is

sfd: *fileid dirid* [*filemode-number*]

If *fileid* has the form *filename filetype*, the filename, filetype, or both can be specified as *, indicating that the filename and/or filetype is not to be considered while reading the directory. If *filemode-number* is specified, only entries for files with the specified mode number are read.

Here are a few examples of **`sfd`**-style filenames for use with **`dirsearch=file`**, **`dirsearch=all`**, or **`dirsearch=allauth`**:

`sfd: * * .`
specifies that entries for all files in the user's top directory are to be read.

`sfd: * exec devel`
specifies that entries for all files with the filetype EXEC in the directory identified by the namedef **`devel`** are to be read.

`sfd: * * mike.backup 2`
specifies that entries for all files with filemode number 2 in the subdirectory BACKUP, belonging to the user MIKE, are to be read.

When you open a directory with **`dirsearch=subdir`**, the pathname specifies only the directory that is to be read. This format of **`sfd:`**-style filename is also used when you call **`remove`**, **`rename`**, **`access`**, **`cmsstat`**, or **`sfsstat`** for a Shared File directory.

Here are a few examples of **`sfd`**-style filenames to use with **`dirsearch=subdir`**:

`sfd: .`
specifies that entries for all subdirectories of the user's top directory are to be read.

`sfd: master`
specifies that entries for all subdirectories of the directory associated with the namedef **`master`** are to be read.

After you open a Shared File System directory, you read it as any other file. The data you read consist of a number of records, one for each matching file if **`dirsearch=subdir`** is not specified, and one for each subdirectory if

dirsearch=subdir is specified. Mappings for the formats of these records are provided in the header file **<cmsstat.h>**. The following are more exact specifications:

dirsearch=file
specifies that the records are mapped by a portion of **struct MAPALL**. Only the first **L_file** bytes of this record are actually read.

dirsearch=all or **dirsearch=allauth**
specifies that the records are mapped by **struct MAPALL**. The number of bytes read is **L_all**.

dirsearch=subdir
specifies that the records are mapped by **struct MAPDIR**. The number of bytes read is **L_dir**.

These structures generally contain binary data. Therefore, if new-line characters appear in the data, you should open **sfd** files for binary rather than text access to avoid possible confusion. Also, none of the seeking functions **fseek**, **ftell**, **fsetpos**, or **fgetpos** are supported for Shared File directories.

Here is a simple example that opens a directory with **dirsearch=file** to print out the names of the files in the directory:

```
#include <lcio.h>
#include <cmsstat.h>

struct MAPALL dir_data;

main() {
   FILE *dir;
   int count = 0;
      /* Open top directory for input. */
   dir = afopen("sfd: * * .", "rb", "", "dirsearch=file");

   if (!dir) abort();
   for(;;) {
      fread(&dir_data, L_file, 1, dir);  /* Read one entry. */
      if (ferror(dir)) abort();
      if(feof(dir)) break;
      if (count == 0)     /* Write title before first line. */
         puts("Files in directory .:");
      ++count;
      printf("%8.8s %8.8s\n", dir_data.file.name,
      dir_data.file.type);
   }

   printf("\n%d files found.\n", count);
   fclose(dir);
   exit(0);
 }
```

Using VSAM Files

VSAM (Virtual Storage Access Method) is a file organization and access method oriented toward large collections of related data. Ordinary MVS and CMS files are organized as a sequential collection of characters, possibly grouped into records. To locate a particular record, it is necessary to read the entire file, until the appropriate record is found. VSAM files are organized according to keys that serve as record identifiers. This makes VSAM especially useful and efficient for many large-scale data processing applications.

For example, suppose you have a file containing a record for each employee in a company. If the data are stored in a normal MVS or CMS file, to update a single record you must read the entire file, until the correct record is found. Alternatively, the data can be stored in a VSAM file, using employee name or employee number as a key. With this organization, a program can immediately read and update any record, given the key value, without having to read the rest of the file.

VSAM files are described in more detail in the IBM publication *MVS/DFP Using Data Sets*. Consult this publication for a more complete description of VSAM files and services.

Kinds of VSAM files

Four kinds of VSAM files exist: KSDS, ESDS, RRDS, and LDS files. Following are the characteristics of these files:

KSDS (Key-Sequenced Data Set)
: is a VSAM file in which each record has a character key stored at some offset in each record. Each record must have a unique key. Records are stored and retrieved in key sequence. They can be added to or deleted from the data set in any order; that is, you can freely add new records between existing records. When records are modified, the length of the record can change, but the value of the key cannot be changed. Most VSAM files are KSDS files.

KSDS files can have *alternate indices*, which are auxiliary VSAM files that provide access to records by a key field other than the primary key. Access to records by an alternate index is accomplished by a path, which is an artificial data set name assigned to the combination of the base KSDS and the alternate index. An alternate index can be defined for a KSDS using a nonunique key field. For example, an employee file cannot have last name as its primary key because more than one employee may have the same last name. But a path to the employee file can use last name as its key, allowing quick access, for instance, to the records for all employees named Brown.

ESDS (Entry-Sequenced Data Set)
: is a VSAM file in which each record is identified by its offset from the start of the file. This offset is called a relative byte address (RBA). When a record is added to the file, it is added to the end, and the RBA of the new record is returned to the program. The RBA thus serves as a logical key for the record. ESDS data sets can have records of different lengths, but when a record is replaced, the length must not be changed. Also, records cannot be deleted.

Like KSDS files, ESDS files can have alternate indices that provide keyed access to the records of the ESDS. For instance, you can choose to make an employee file an ESDS file, with the records arranged by the order in which they were added to the file. You can still access records in this file using last names by building an alternate index with last name as the key over the ESDS. The same rules apply when you use a path to access an ESDS as when you access the ESDS directly; that is, you cannot change the length of a record or delete a record. Except for these restrictions, a path over an ESDS is treated as a KSDS because records accessed through the path are always arranged in the order of the alternate keys.

RRDS (Relative-Record Data Set)
: is a VSAM file in which each record is identified by record number. Records can be added or deleted in any order, and the file can have holes; that is, it is not necessary that all possible record numbers be defined. Records in an RRDS file are normally all the same length; SAS/C does not support RRDS files with variable-length records. Alternate indices over an RRDS are not supported.

LDS (Linear-Data Set)
: is a VSAM file consisting solely of a sequence of characters divided into 4096-byte pages. Unlike other VSAM files, LDS files are not keyed, and normally are accessed via the supervisor DIV (Data In Virtual) service, as described in the IBM publication *MVS/ESA Application Development Guide.* LDS files are supported only under MVS.

Access to VSAM files using standard I/O

You can access all kinds of VSAM files using the standard C I/O functions defined by **`<stdio.h>`**. Because of the special characteristics of VSAM files, there are restrictions for some types of VSAM files:

- □ Access to a KSDS file via standard I/O must be for input only. Output cannot be supported, because records are written in key order, and the C standard specifies that new data can be written only to the end of a file. That is, it is not possible, using standard I/O, to insert characters between existing characters of an old file. When you read a KSDS file using standard I/O, the records are always retrieved in ascending key order.
- □ Access to an ESDS file via standard I/O is fully supported, in both text and binary mode.
- □ Access to an RRDS file via standard I/O is supported only in binary mode. An RRDS file is processed using the **''rel''** access method. This means that you can use the **fseek** and **ftell** functions, which provides compatibility with file behavior on UNIX operating systems.
- □ Access to an LDS file via standard I/O is supported only in binary mode. The library uses the DIV macro to access the file rather than reading or writing it directly. An LDS file is processed using the **''rel''** access method. This means that the **fseek** and **ftell** functions can be used, which provides compatibility with file behavior on UNIX operating systems. See "Advanced MVS I/O Facilities" on page 3-64 for more information on accessing LDS files.

Keyed access to VSAM files

Because standard C I/O assumes that files are simply sequences of characters stored at unchanging offsets in the file, standard C is not suited to exploiting VSAM capabilities. For this reason, the SAS/C library provides a keyed-access mode for VSAM files designed to take advantage of their unique properties. Keyed access is an alternative to text or binary access, specified by the open mode argument to the **fopen** or **afopen** function, as shown in the following statements:

```
ftext = fopen("ddn:ESDS", "r+");   /* Open ESDS for text access.   */
fbin  = fopen("ddn:ESDS", "r+b");  /* Open ESDS for binary access. */
fkey  = fopen("ddn:ESDS", "r+k");  /* Open ESDS for keyed access.  */
```

Only a subset of the standard I/O functions (shown in the following list) are available for files opened for keyed access; that is, this list shows the functions that make sense for such files.

afflush	flush output buffers, synchronize file access
clearerr	reset previous error condition
clrerr	reset previous error condition
fattr	return file attributes
fclose	close file
feof	test for end of file

`ferror`	test for error
`ffixed`	test for fixed-length records
`fnm`	return filename
`fterm`	test whether file is the terminal
`setbuf`	change buffering (null operation)
`setvbuf`	change buffering (null operation).

The following keyed-access functions are supported only for VSAM files. These functions are described in more detail later in this section:

`kdelete`	delete the last record retrieved
`kgetpos`	return position of current record
`kinsert`	add a new record
`kreplace`	replace the last record retrieved
`kretrv`	retrieve a record
`ksearch`	locate a record
`kseek`	reposition keyed file
`ktell`	return RBA of current record.

Keyed access is not supported for VSAM LDS files because these files are not divided into records and have no keys.

Records and keys

Keyed-access functions for VSAM process one record at a time, rather than one character at a time. Most functions have arguments that are pointers to records or pointers to keys. Because C is not, in general, a record-oriented language, you need to be careful when defining data structures for use with VSAM.

Many VSAM files have fixed-length records, where all records have the same format. These files are easy to process in C, because the record can be represented simply as a structure, as shown in this simple example:

```
struct {
   char name[20];
   char address[50];
   short age;
}  employee;

kretrv(&employee, NULL, 0, f);
```

This example reads records of a single type from a VSAM file. More complicated files may have records with different lengths or types; C unions can be helpful in processing such files:

```
struct personal {
   char name[20];
   char rectype;     /* = P for personal */
   char address[50];
   short age;
} ;
```

```
struct job {
   char name[20];
   char rectype;          /* = J for job */
   long salary;
   short year_hired;
}
union {
   struct personal p_rec;
   struct job j_rec;
}  employee;

kretrv(&employee, NULL, 0, f);
```

The call to the **kretrv** function can read a record of either type; then the **rectype** field can be tested to determine which type of record was read. Here is an example showing the replacement of a record with several types:

```
if(employee.p_rec.rectype == 'P')
   recsize = sizeof(struct personal);
else recsize = sizeof(struct job);
kreplace(&employee, recsize, 0, f);
```

If the length were specified as **sizeof(employee)** and the record were a job record, more data would be written than defined in the record, and file space is wasted.

Any characters can occur in a record. Neither new-line characters ('\n') nor null characters ('\0') have any significance.

For a KSDS file, a record key is always a fixed-length field of the record. Any characters can appear in a key, including the new-line character ('\n') and the null character ('\0'). Also, the key is not restricted to being a character type; for some files, the key might be a **long**, a **double**, or even a structure type.

When you have character keys, you should be sure to specify all characters of the key. For instance, consider the following call to the **ksearch** function, intended to retrieve the record whose key is Brown in the employee file described previously:

```
ksearch("Brown", 0, K_exact, f);
```

This search will probably fail, because the key length for this file is 20 characters. The **ksearch** function looks for a record whose key is the characters **''Brown''**, followed by a null character, followed by 14 random characters (whatever comes after the string **''Brown''** in memory), and probably will not find one.

Usually, strings in VSAM files are padded with blanks, so the following example shows the correct usage:

```
char key[20];
memcpyp(key, "Brown", 20, 5, ' ');  /* Copy key and blank pad. */
ksearch(key, 0, K_exact, f);
```

ESDS and RRDS files do not have physically recorded keys. However, the RBA (for an ESDS) and the record number (for an RRDS) serve as logical keys for these files. The structures representing these records in a C program must include an **unsigned int** or **unsigned long** field at the start of the record to hold the key value. This key is not actually recorded in the file. In this example, record 46 is inserted into an RRDS:

```
struct {
   unsigned long recno;
   char name[20];
   char address[60];
} rrds_rec;

rrds_rec.recno = 46;
kinsert(&rrds_rec, sizeof(rrds_rec), NULL, rrds);
```

The key (46) is specified in the first 4 bytes of the record. Note that the key is not actually stored in the file. The size of the C record is 84 characters, but the length of the record in the VSAM file is 80 characters because the key is not physically recorded.

Rules for keyed access

The keyed-access functions are record-oriented rather than character-oriented. When a keyed file is used, it can be in one of the following states:

- Immediately after the file is opened, there is no current record defined. This means that functions such as **kdelete** and **kreplace**, which affect the current record, are not allowed at this point. After successful use of the **kinsert**, **kreplace**, **kdelete**, **ksearch**, or **kseek**, the file is also in this state.
- After successful use of **kretrv**, the retrieved record becomes current. This means that the record can be updated or deleted and that its address can be obtained with the **ktell** or **kgetpos** functions. The current record can either be held for update or not held. The record is not held if the file is opened for input only or if **K_noupdate** was specified as an argument to the **kretrv** call. Otherwise, the record is held. Replacement or deletion of a record is allowed only if it is held for update. Additionally, some other VSAM processing is different, depending on whether the current record is held. For more information, see "VSAM pitfalls" on page 3-79.
- After an error in processing, the file is in an error state. You need to call the **clearerr** function to continue use of the file. In many cases after an error, the file position is undefined, and you have to use **ksearch** or **kseek** to reposition the file before continuing.

Unlike some other kinds of files, your program can open the same VSAM file more than once. The same file can be opened any number of times, either using the same filename, or using different names. A file can also be opened via several paths. The open modes do not need to be the same in this case. For example, one stream can access the file for input only, and another can access to append. However, each opening of the file must specify keyed access; that is, standard I/O and keyed I/O to the same file cannot be mixed.

When several streams access the same VSAM file, only one of them can hold a particular record for update. If one stream attempts to retrieve a record held by another stream and **K_noupdate** is not specified, retrieval will fail. Because of the way that VSAM holds records for update, it is also possible for two streams accessing the same file to interfere with each other's processing of different records. See "VSAM pitfalls" on page 3-79 for more information.

Using a KSDS

The following are considerations unique to processing a KSDS:

- Records can be retrieved in either ascending or descending key order.
- KSDS files support many kinds of searches. You can search for a record with a particular key, a record with a matching partial (or generic) key, or a record with a key greater than or equal to a particular value. You can search either forward or backward to optimize retrieval in the chosen direction. (A backward search for an inexact key finds the last record whose key is less than or equal to the specified value.) Backward searches are restricted in paths that allow duplicate keys. See the **`ksearch`** function description for further details.
- Records can be added to a KSDS at any point in the file. After a record is added, the file is positioned immediately following the new record.
- Records can be deleted from a KSDS. After a record is deleted, the file is positioned to the record following the deleted record. Records cannot be deleted from a path whose base is an ESDS.

Using an ESDS

The following are considerations unique to processing an ESDS:

- ESDS files have no physical keys. A record's RBA is used as a key by the C library routines. The areas used for input or output of ESDS records must include 4 bytes at the start for the record's RBA, in addition to the data stored in the file.
- Records can be retrieved in either ascending or descending RBA order.
- Only exact searches are allowed; that is, you can locate a record with a specific RBA, but it is not possible to search for a record whose RBA is greater than or less than a specific value.
- New records are always inserted at the end of file in an ESDS. The RBA of the new record is optionally stored by **`kinsert`** and is not generally predictable before the record is inserted.
- Deletion of records from an ESDS is not supported. Similarly, it is not permitted to change the length of an existing record.

Using an RRDS

The following considerations are unique to processing an RRDS:

- RRDS files have no physical keys. A record's record number is used as a key by the C library routines. The areas used for input or output of RRDS records must include 4 bytes at the start for the record number, besides the data stored in the file.
- Records can be retrieved in either ascending or descending record number order. Records that are not defined are skipped; that is, if records 1 and 3 have been stored, but record 2 is not defined (or has been deleted), record 3 will be retrieved after record 1, and no error will occur due to the missing record.
- An RRDS does not support partial or generic key searches. However, you can search for the first record whose number is greater than or less than a specific value.
- New records can be inserted anywhere in an RRDS where a record does not exist already. After a record is inserted, the file is positioned to the record number following the insertion.
- Records can be deleted from an RRDS. After a deletion, the file is positioned to the record number after the deleted record.
- SAS/C does not support RRDS data sets that contain varying-length records.

Using an alternate path

The following are considerations unique to processing an alternate path. An alternate path to a KSDS or an ESDS is treated, in general, as a KSDS, except that some operations are forbidden for a path whose base is an ESDS:

- Records can be retrieved in either ascending or descending key order. If the file is a path with duplicate keys, records with the same key are retrieved in the same order that they were added to the file, whether retrieval is forward or backward. When processing a path with duplicate keys, you cannot switch between backward and forward retrieval without performing a search.
- In general, paths support the same kinds of searches as KSDS files. You can search for a record with a particular key, a record with a matching partial (or generic) key, or a record with a key greater than or equal to a particular value. You can search either forward or backward to optimize retrieval in the chosen direction. (A backward search for an inexact key finds the last record whose key is less than or equal to the specified value.) Backward searches are restricted in paths that allow duplicate keys. See the **ksearch** function description for further details.
- Records can be added to a path at any point in the file. After a record is added, the file is positioned immediately following the new record. Even for a path that allows duplicate keys, you are not permitted to add a record with the same primary key as an existing record.
- Records can be replaced via an alternate path. Neither the primary key nor the alternate key can be changed. If the base of the path is an ESDS, the record length cannot be changed.
- Records can be deleted from a path whose base is a KSDS. After a record is deleted, the file is positioned to the record following the deleted record. Records cannot be deleted from a path whose base is an ESDS.

VSAM pitfalls

The VSAM access method is highly optimized for performance when processing large files. Sometimes this optimization can produce incorrect results for programs that process the same VSAM data set using more than one FILE or more than one path. This section describes some of these situations and suggests how to circumvent them. These pitfalls apply only to programs that access the same file in several ways. Simple VSAM programs that open each VSAM file only once are not affected.

Note: Sharing of VSAM files between users through SHAREOPTIONS(3) or the SHAREOPTIONS(4) Access Method Services (AMS) option is not supported by the SAS/C library. If you ignore this restriction, lost records, duplicate records, or file damage may occur.

When the same file is accessed through several paths, a problem can occur when VSAM attempts to avoid reading a record from disk because a copy exists in memory. If a request is made to read a record and the record is not to be updated, VSAM may return an obsolete copy of the record from memory to the program, rather than reading a current copy from disk. If this is a problem for your application, you can always retrieve records for update by not specifying **K_noupdate**, regardless of whether you intend to update them. This ensures that you get the most recent version of a record at the cost of additional disk I/O. But note that you cannot retrieve a record for update if you open the file with open mode **'rk'**. Alternately, you can use the **afflush** function to flush all buffers in memory. After using **afflush**, retrievals access the most recent data from disk, because there is no copy in memory.

Many programs cannot be affected by this problem. For example, if your program processes records in key order, it probably does not ever attempt to retrieve a record after it has been updated. For such a program, there is no need to avoid the use of **K_noupdate** on retrieval.

Another potential problem has to do with the way that VSAM stores records. Records in VSAM files are organized into control intervals of a fixed size. Each disk access consists of the reading or writing of an entire control interval. When VSAM is said to be holding a record, the control interval is actually what is held. This means that an attempt to read a record for update using one stream may fail, because another record in the same control interval is held by another stream. In general, when this may occur cannot be predicted, although records whose keys are close to each other are more prone to this condition.

This condition can be recognized by the setting of the value `EINUSE` for the `errno` variable. Resolving the condition is more difficult. It is possible to release a hold on a record without updating the record by the call `kseek(fp, SEEK_CUR)`. This call does not change the file position, which means that `kretrv` can be used to retrieve the next record, as if `kseek` had never been called. Also, sometimes you may be able to organize your program so that it normally retrieves records using `K_noupdate` and, if the program decides that the record should be modified, retrieves the record a second time for update immediately before replacing or deleting it.

VSAM-related amparms

For some VSAM files, processing performance can be improved by allocating storage for additional I/O buffers. SAS/C allows you to specify buffer allocation for VSAM files using the following amparms. (Note that the `order` amparm can also have an effect on performance.

- The `bufnd` amparm specifies the number of I/O buffers VSAM is to use for the data records. This option is meaningful only for VSAM files and is equivalent to coding a BUFND value on an ACB assembler macro used to open the VSAM file. A data buffer is the size of a control interval in the data component of a VSAM cluster. For keyed access and `order=random`, the `bufnd` default is 2. For standard I/O VSAM access or keyed access with `order=seq`, the `bufnd` default is 4. Generally, with sequential access, the optimum value for the data buffers is six tracks, or the size of the control area, whichever is less. For skip-sequential processing, specifying two tracks for the data buffers is a good starting point. Specification of a `bufnd` value larger than the default generally yields performance improvements for applications that primarily do sequential input processing of the VSAM file or initial loading (sequential writes after first open) for a VSAM file opened with keyed access. In other situations, specifying a larger `bufnd` value may yield no improvement, or may actually degrade performance by tying up large amounts of virtual storage and causing excessive paging.
- The `bufni` amparm specifies the number of I/O buffers VSAM is to use for index records. This option is meaningful only for VSAM KSDS files and is equivalent to coding a `BUFNI` value on an ACB assembler macro used to open the VSAM file. An index buffer is the size of a control interval in the index component of a keyed VSAM cluster. For keyed access (random or skip sequential), `bufni` defaults to 4, and for text or binary access (mainly sequential), it defaults to 1. For keyed access other than initial load, the optimum `bufni` specification is the number of high-level (nonsequence set) index buffers + 1. You can determine this number by subtracting the number of data control areas from the total number of index control intervals within the dataset. You can use an upperbound `bufni` specification of 32, which accommodates most VSAM files with reasonable index control interval and data control area sizes (cylinder allocated data component) up to the four-gigabyte maximum data component size allowed. Large `bufni` specifications incur little or no performance penalty, unless they are excessive.
- The `bufsp` amparm specifies the maximum number of bytes of storage to be used by VSAM for all I/O buffers. This option is meaningful only for VSAM files and is equivalent to coding a `BUFSP` value on an ACB assembler macro used to open the file. A data or index buffer is the size of a control interval in the data or index

component. For a valid **bufsp** specification (minimum of one index and two data buffers), VSAM allocates data and index buffers as follows:

- For **order=seq** amparm, initial keyed access load, or text or binary access, one or two index buffers are allocated, and the remaining bytes are allocated to data buffers.
- For keyed access and **order=random**, two data buffers are allocated, and the remaining bytes are used for index buffers.

A valid **bufsp** specification generally overrides any **bufnd** or **bufni** specification. However, the VSAM rules for doing this are fairly complex, and you should consult the appropriate IBM VSAM Macro Reference manual for your system for more information on the ACB macro BUFSP option.

VSAM I/O Example

This example consists of three pieces of code. The first piece is the SAS/C VSAM example program; the second piece is the JCL used to create, load, and update the VSAM file; and the third piece is the JCL used to compile and link the VSAM example.

The SAS/C VSAM example program, **KSDS**, demonstrates how to load, update, search, retrieve, and delete records from a KSDS VSAM file. Two VSAM files are used:

ddn:ITEM	the VSAM file being used
ddn:DATA	where records for initially loading the VSAM file are stored

Two data files are used:

ddn:UPDATE	contains the records for loading and updating
ddn:DELETE	contains the keys for the records being deleted

```
#include <stdio.h>
#include <lcio.h>
#include <fcntl.h>

#define BUFSIZE 80
#define VBUFSIZE 50
#define KEYSIZE 19

void loadit(void);          /* Load a VSAM file.                   */
void update(void);          /* Update a VSAM file.                 */
void printfil(void);        /* Print a VSAM file.                  */
void add_rep(void);         /* Add or update specific records.     */
void del_rec(void);         /* Delete specific records.            */

FILE *vfptr1,               /* ptr to the VSAM file                */
     *vfptr2,               /* another ptr to the VSAM file        */
     *fptr;                 /* ptr to the DATA file                */

char buffer[BUFSIZE+1];     /* buffer for reading input file data */
char vbuffer[VBUFSIZE+1];   /* VSAM record buffer                  */
char key[KEYSIZE+1];        /* key field for VSAM record           */
```

```
main()
{
   unsigned long offset;
      /* If VSAM file has been loaded, update.  Otherwise LOAD.    */
   puts("Has the VSAM file been loaded?");
   quiet(1);
      /* Attempt to open the file r+k.  If that works, then it     */
      /*  is loaded.  The open could fail for reasons other than   */
      /*  that the file has not yet been loaded.  In this case,    */
      /*  loadit will also fail, and a library diagnostic will     */
      /*  be printed.                                              */
   vfptr1 = afopen("ddn:ITEM", "r+k", "", "keylen=19, keyoff=0");
   quiet(0);

   if (!vfptr1){
      puts("File has not been loaded. Load it.");
      loadit();
      }
   else{
      puts("File has been loaded. Update it.");
      update();
      }

      /* Show the current state of the VSAM file. */
   printfil();

      /* Est. 2nd ptr to VSAM file.  */
      /*Search and print specific records. */
   if ((vfptr2 = afopen("ddn:ITEM", "r+k", "", "keylen=19, keyoff=0"))
      == NULL){
      puts("Could not open VSAM file a 2nd time");
      exit(99);
      }
      /* Search for some specific records by key. */
   puts("\n\nDo some searching");
   memcpy(key, "CHEMICAL FUEL      ", KEYSIZE);
   key[KEYSIZE]='\0';       /* Terminate the key. */
   printf("Search for %s\n", key);
   ksearch(key, 0, K_noupdate | K_exact, vfptr1);
   memcpy(key, "HOUSEHOLD PAN      ", KEYSIZE);
   key[KEYSIZE]='\0';       /* Terminate the key. */
   printf("Now Search for %s\n", key);
   ksearch(key, 0, K_noupdate | K_exact, vfptr2);

      /* Retrieve the records found. */
   puts("nnOK, now retrieve the records that we found");
   kretrv(vbuffer, NULL, K_noupdate, vfptr1);
   vbuffer[VBUFSIZE]='\0';
   puts(vbuffer);
   kretrv(vbuffer, NULL, K_noupdate, vfptr2);
   vbuffer[VBUFSIZE]='\0';
   puts(vbuffer);

   fclose(vfptr2);
```

```
      /* Find the first and last records in the file and their RBA. */
   kseek(vfptr1, SEEK_SET);
   kretrv(vbuffer, NULL, K_noupdate, vfptr1);
   vbuffer[VBUFSIZE]='\0';
   printf("\nThe first record in the file is:\n%s\n", vbuffer);
   offset = ktell(vfptr1);
   printf("Its RBA is: %lu\n", offset);
   kseek(vfptr1, SEEK_END);
   kretrv(vbuffer, NULL, K_backwards | K_noupdate, vfptr1);
   vbuffer[VBUFSIZE]='\0';
   printf("\nThe last record in the file is:\n%s\n", vbuffer);
   offset = ktell(vfptr1);
   printf("Its RBA is: %lu\n", offset);
}

   /* This is the loadit function, which does the initial */
   /*  loading of the VSAM file.                          */
void loadit()
{

   puts("Loading the VSAM file");
   if ((fptr = fopen("ddn:DATA", "rb")) == NULL){
      puts("Input file could not be opened");
      return;
   }

      /* We must attempt to open the file again.  Since we are here, */
      /*  we know that the first attempt failed, probably because    */
      /*  the file was empty.                                        */

   if ((vfptr1 = afopen("ddn:ITEM", "a+k", "", "keylen=19, keyoff=0"))
      == NULL){
      puts("VSAM file could not be opened");
      return;
   }

   while (afread(buffer, 1, BUFSIZE, fptr)){
      kinsert(buffer, VBUFSIZE, NULL, vfptr1);
   }
}


/* The following function updates the VSAM file by calling functions */
    to add, replace, and delete records.                              */

void update()
{
   puts("Updating the VSAM file");
   printfil();
   add_rep();
   del_rec();
}
```

```
   /* The add_rep function updates the VSAM file by adding or  */
   /*  replacing records specified in DDN:UPDATE.              */

void add_rep()
{

   puts("\nUpdating specified VSAM records");
   if ((fptr = fopen("ddn:UPDATE", "rb")) == NULL){
      puts("Update file could not be opened");
      return;
   }
   puts("\n");

      /* Search VSAM file for records whose keys match those in */
      /*  UPDATE file. If a match is found, update record.      */
      /*  Otherwise, add record.                                */

   kseek(vfptr1, SEEK_SET);
   while (afread(buffer, 1, BUFSIZE, fptr)){
      memcpy(key, buffer, KEYSIZE);
      if ((ksearch(key, 0, K_exact, vfptr1)) > 0){
        kretrv(vbuffer, NULL, 0, vfptr1);
        vbuffer[VBUFSIZE]='\0';
        printf("Replace the record:\n%s\n", vbuffer);
        memcpy(vbuffer, buffer, VBUFSIZE);
        vbuffer[VBUFSIZE]='\0';
        printf("With:\n%s\n", vbuffer);
        kreplace(vbuffer, VBUFSIZE, vfptr1);
      }
      else{
        memcpy(vbuffer, buffer, VBUFSIZE);
        vbuffer[VBUFSIZE]='\0';
        printf("Can't find this record, so we'll add it:\n%s\n",vbuffer);
        kinsert(vbuffer, VBUFSIZE, NULL, vfptr1);
      }
   }
   fclose(fptr);
}

   /* The del_rec function deletes VSAM records               */
   /* with specified keys.                                    */

void del_rec()
{
   puts("\nDeleting specified VSAM records");
   if ((fptr = fopen("ddn:DELETE", "rb")) == NULL){
      puts("Delete file could not be opened");
      return;
   }

      /* Search VSAM file for records whose keys match those in */
      /*  DELETE file. If a match is found, delete record.      */
      /*  Otherwise, issue a message that no match was found.   */
   kseek(vfptr1, SEEK_SET);
   while (afread(buffer, 1, BUFSIZE, fptr)){
      memcpy(key, buffer, KEYSIZE);
      key[KEYSIZE]='\0';
      if ((ksearch(key, 0, K_exact, vfptr1)) > 0){
        kretrv(vbuffer, NULL, 0, vfptr1);
        vbuffer[VBUFSIZE]='\0';
        printf("Delete the record:\n%s\n", vbuffer);
        kdelete(NULL, vfptr1);
      }
      else
        printf("Couldn't find a record with the key: %s\n", key);
   }
   fclose(fptr);
}


   /* The function printfil prints the contents of the VSAM file. */
                                |
void printfil()
{
   int i=0;

   puts("\n\nHere is the current state of the VSAM file");
   puts("   ITEM             QTY SZ BIN DESC    COMMENTS  ");
   kseek(vfptr1, SEEK_SET);
   while ((kretrv(buffer, NULL, K_noupdate, vfptr1)!=-1) &&
         !feof(vfptr1)){
      buffer(|VBUFSIZE|) = '\0';
      printf("%d %s\n", i, buffer);
      i++;
   }
}
```

The JCL file KSDSGO creates a KSDS VSAM file, loads it using the above program (**KSDS**), and then updates it using **KSDS**.

Note: The DEFINEIT step will fail the first time if there is no VSAM file to delete. You must either comment out the DELETE step the first time or create a dummy VSAM file that can be deleted before this job is run:

```
//*-----------------------------------------------------------------
//* DEFINE THE VSAM KSDS
//*-----------------------------------------------------------------
//DEFINEIT EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
 DELETE (yourid.ksds.vsamfile) PURGE CLUSTER
 DEFINE CLUSTER (NAME(yourid.ksds.vsamfile) INDEXED VOLUMES(YOURVOL) -
             TRACKS(2 2) KEYS(19 0) RECORDSIZE(50 100)            -
             FREESPACE(0 0) CISZ(512) )                           -
       DATA  (NAME(yourid.ksds.vsamfile.DATA))                    -
       INDEX (NAME(yourid.ksds.vsamfile.INDEX))
 LISTCAT ENTRIES(yourid.ksds.vsamfile) ALL
```

```
/*
//*-------------------------------------------------------------------
//* LOAD THE VSAM KSDS USING SAS/C
//*-------------------------------------------------------------------
//LOADIT EXEC PGM=KSDS
//STEPLIB DD DSN=your.load.dataset,DISP=SHR
//        DD DSN=sasc.transient.library,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSTERM  DD SYSOUT=A
//SYSUDUMP DD SYSOUT=A
//ITEM     DD DSN=yourid.ksds.vsamfile,DISP=SHR
//DATA     DD  *
AUTO STARTER       99 02 92  WARRANTY  MODERATE
CHEMICAL ACID      05 04 00  PH10      MODERATE
CHEMICAL EAAAAA    55 75 50  ALCOHOL   CHEAP
CHEMICAL FUEL      45 77 80  DIESEL    EXPENSIVE
CHEMICAL GAS       10 30 50  LEADED    CHEAP
HOUSEHOLD PAN      03 10 33  METAL     CHEAP
/*
//*
//*-------------------------------------------------------------------
//* OBTAIN AN IDCAMS LISTCAT
//*-------------------------------------------------------------------
//IDCAMS3  EXEC PGM=IDCAMS
//SYSPRINT DD SYSOUT=A
//SYSIN DD *
  LISTCAT ENTRIES(yourid.ksds.vsamfile) ALL
/*
//*
//*-------------------------------------------------------------------
//* ADD/UPDATE/DELETE RECORDS TO THE VSAM KSDS
//*-------------------------------------------------------------------
//UPDATE EXEC PGM=KSDS
//STEPLIB DD DSN=your.load.dataset,DISP=SHR
//        DD DSN=sasc.transient.library,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSTERM  DD SYSOUT=A
//SYSUDUMP DD SYSOUT=A
//ITEM     DD DSN=yourid.ksds.vsamfile,DISP=SHR
//UPDATE   DD  *           LIST OF RECORDS TO BE ADDED OR UPDATED
CHEMICAL FUEL      45 77 80  GASOLINE  CHEAP
CHEMICAL FUELS     55 67 81  PROPANE   CHEAP
/*
//DELETE   DD  *           LIST OF KEYS OF RECORDS TO BE DELETED
AUTO STARTER
BOGUS ID
/*
//
```

The JCL file called KSDSCL compiles and links **`KSDS`**.

```
//*-----------------------------------------------------------------
//* COMPILE AND LINK THE PROGRAM
//*-----------------------------------------------------------------
//CL EXEC LC370CL
//C.SYSLIN DD DSN=your.obj.dataset(KSDS),DISP=OLD
//C.SYSIN  DD DSN=your.source.dataset(KSDS),DISP=SHR
//LKED.SYSLIN DD DSN=your.obj.dataset(KSDS),DISP=OLD
//LKED.SYSLMOD DD DSN=your.load.dataset(KSDS),DISP=SHR
//
```

SAS/C I/O Questions and Answers

The following are frequently asked questions about SAS/C I/O:

Flushing output to disk

Q. My program runs for days at a time. I want to be sure that all data I write to my files are actually stored on disk in case the system fails while the program is running. **`fflush`** does not seem to guarantee this. What can I do?

A. It is true that using **`fflush`** on a file does not guarantee that output data are immediately written to disk. For a file accessed as a binary stream, **`fflush`** passes the current output buffer to the system I/O routines, but there is no guarantee that the data are immediately transmitted. (For instance, under MVS the data are not written until a complete block of data has been accumulated.) This situation is not limited to MVS and CMS. For instance, **`fflush`** under most versions of UNIX simply transfers the data to a system data buffer, and there is no guarantee that the data are immediately written to disk.

Even after output data are physically written to disk, they may be inaccessible after a system failure. This is due to the details of the way that disk space is managed under MVS and CMS. For instance, under CMS, the master file directory for a minidisk maps the blocks of data associated with each file. If a program adds new data blocks to a file, but the system fails before the master file directory is updated, the new blocks are inaccessible. CMS only updates the master file directory when a command terminates, or when the last open file on the disk is closed by CMS. A similar problem occurs under MVS with the end-of-file pointer in the data set label, which is updated only when the file is closed.

The recommended solution to this problem is the nonportable SAS/C function **`afflush`**. This function is similar to **`fflush`** but guarantees that all data has been written to disk and that the master file directory or VTOC has been updated to contain current information. For programs using UNIX style I/O, the function **`fsync`** can be used in the same way.

Comparing C standard I/O to other languages' I/O

Q. I have compared a small C program with a small COBOL program. Both programs simply copy records from one file to another. Why is C standard I/O so much slower than COBOL, and is there anything I can do about it?

A. A simple COBOL file copy and a simple C file copy are not comparable. COBOL I/O can be implemented much more efficiently on the 370 than C standard I/O because the COBOL I/O model is a record I/O model, corresponding closely to the I/O model implemented by the operating system. COBOL clauses such as "RECORD CONTAINS 80 CHARACTERS" and "BLOCK CONTAINS 40 RECORDS" allow the compiler to generate code that invokes the operating system's access methods directly.

C standard I/O, on the other hand, is stream-oriented. The C library cannot call the access method directly because of the need to support text file translation and complicated interfaces like **`ungetc`**, **`fflush`**, and **`fseek`**. Even if the program does not use these capabilities, the library must still support them. In addition, because of the requirement that compiled code be system independent, and because the meanings of attributes like **`reclen`** and **`blksize`** differ from MVS to CMS, C I/O cannot be optimized based on knowledge of these attributes, as COBOL can.

The SAS/C OS and CMS low-level I/O functions enable you to access files at the same low level used by COBOL. When you use low-level I/O in C, you will find performance similar to that of COBOL. If you do not want to use low-level I/O, then refer to the next question in this section.

Note: The discussion here applies when you compare C I/O to other languages such as PL/I. However, the difference is not as great because these languages also compare unfavorably to COBOL, and for the same reason: their I/O models are not as close to the 370 model as COBOL's, although they are closer than the C model.

Efficient I/O

Q. What can I do to improve the performance of I/O?

A. Here are some recommendations for improving the performance of I/O. Each recommendation should be evaluated individually because many are not relevant to every application:

- Avoid UNIX style I/O if possible, except when using the OpenEdition hierarchical file system. If you need UNIX I/O properties, such as byte-addressability, use files suitable for "rel" access to avoid the overhead of copying the data to a temporary file.
- Use binary rather than text access, if you have a choice. When you read or write a file as a text stream, every character read or written must be inspected to see if it is a new-line character and, therefore, requires special treatment. No such tests are necessary with binary access. If your application uses **`fgets`** or **`fputs`** to process data one line at a time and it is not required to be portable, investigate whether it could be changed to use **`afread`** or **`afwrite`** instead.
- Avoid doing I/O one character at a time. Especially avoid the **`fgetc`** and **`fputc`** functions. The C standard requires that these functions generate an actual function call, which introduces a substantial overhead for each character read or written. If you must do I/O one character at a time, use **`getc`** and **`putc`**, which generate inline code and cause a subroutine call only when necessary to read or write a new buffer. For debugged programs that use **`getc`** and **`putc`**, you can **`#define`** the symbol **`_FASTIO`** where appropriate. This increases the speed of **`getc`** and **`putc`** by removing checks for invalid **`FILE`** pointers.
- Read or write an entire record at a time using the multiple item feature of **`fread`** and **`fwrite`** or **`afread`** and **`afwrite`**. Reading less than one record at a time increases the number of subroutine calls required and, therefore, decreases performance. Reading more than one record at a time is not harmful, but it is not particularly beneficial, because data are buffered within the library one record at a time.
- Use a large block size. Data are transferred to and from the disk in blocks, so increasing the block size decreases the number of I/O operations and subroutine calls required to read a given amount of data.
- Use VIO for temporary files under MVS. VIO uses system paging to manage the data and is substantially more efficient than a real temporary data set.

- □ Consider alternatives to `printf`. `printf` is one of the most expensive routines at run time, because of the need to interpret the format string. In many cases, you can use a faster routine to do the same thing. For instance, use `puts(`*`str`*`)` instead of `printf(''%s\n'', str)`.
- □ Put the transient library into MVS LPALIB or a VM shared segment. This has two advantages: it cuts down the overhead associated with dynamically loading I/O routines, and it decreases system paging because all C programs can share the same copy of the library routines.

Processing character control characters as data

Q. How can I process the carriage control characters as data in a file defined as RECFM=*xxA* (where *xx* is F, FB, V, or VB)?

A. Process the file using binary I/O. When you process a record format A file with text I/O, the library manages the carriage control for you, so it can correctly handle C control characters like `'\f'` and `'\r'`. But when you use binary I/O, the library leaves the data alone, so you can process the file as you see fit.

Processing SMF records

Q. How can I best process SMF records using C I/O?

A. The library functions `afreadh` and `afread` are well-suited to reading SMF records. Simple SMF records consist of a header portion containing data items common to all records, including a record type, followed by data whose format is record-type dependent. Complex SMF records may contain subrecords that occur a varying number of times. For instance, a type 30 record contains an I/O subrecord (or section) for each DD statement opened by a job. To process a simple SMF record in C, use `afreadh` to read the common record header, and then use `afread` to read the remainder. The length specified for the call to `afread` is the largest length possible for the record type. (`afread` returns the amount of data actually read.)

To process a complex SMF record in C, use `afreadh` to read each section of the record, using information from previous sections to allow you to map the record. For instance, if the record header indicates you are reading a type 30 record, then you would call `afreadh` again to read the common header for a type 30 record. This header may indicate you have three sections of type A and two of type B. You can then call `afreadh` three times to read the A sections, and two more times to read the B sections.

Note: Using `afread` to read any of the nonheader information is not necessary.

Compatibility between MVS and CMS

Q. What can I do to make my I/O portable between MVS and CMS?

A. You hardly need to do anything special at all. The main source of I/O incompatibility between MVS and CMS is filename syntax. By default, filenames are interpreted as DDnames under MVS, but as CMS disk filenames under CMS. Furthermore, CMS and MVS naming conventions are different. Here are three possible strategies for solving this particular problem (there may be more):

- □ Use DDnames as filenames under both MVS and CMS, so that CMS users have to use the FILEDEF command to define DDnames before running your program. You may use an EXEC to call the program under CMS, in which case the FILEDEFs can be issued by the EXEC.

- □ Use `tso` or `cms` style filenames under both systems, and only use names that are acceptable under both systems. For example, use `tso:config.data`, not `tso:config.user.data`, which is unacceptable under CMS. Note that this may limit your program to being used from TSO under MVS.
- □ Use the `sysname` and `envname` functions to determine at run time whether you are running under MVS or CMS and choose filenames accordingly. This is the most flexible solution because you can choose the filenames most appropriate for each system independently.

After you have solved the filename problem, you will find that your I/O applications move effortlessly between CMS and MVS.

File creation

Q. I call the `creat` function, followed by `close`, to create a file without putting any data in it. But when I open it read-only later, I get a message that the file doesn't exist. What is wrong?

A. You attempted to create an empty file; that is, one containing no characters. CMS does not permit such files to exist. Additionally, for reasons explained in detail in "Technical Background" on page 3-2 , under MVS, empty files with sequential organization are treated by the library as not existing. The ISO/ANSI C Standard permits this interpretation because of the existence of systems like CMS.

You can avoid this restriction in two ways:

- □ You can write a single character to the file (for instance, a single `'\0'`) and ignore this character when you read the file later.
- □ Under MVS, investigate using a PDS member instead of a sequential file, as the restriction does not apply to PDS members. Because there are other restrictions for use of PDS members (such as not being able to add to the end of file), this solution is not feasible for some programs.

Diagnostic Messages

Q. I do not want the library to issue diagnostic messages when I/O errors occur, because my application has complete error-checking code. How do I suppress the library messages?

A. You can use the `quiet` function to suppress all library diagnostics or to suppress diagnostics at particular points in execution. If you use `quiet`, you may occasionally run into errors whose cause cannot be immediately determined. When this happens, you can use the `=warning` run-time option to override `quiet` and obtain a library diagnostic without having to recompile the program.

Converting an Assembler VSAM Application

Q. I'm converting an assembler VSAM application to SAS/C, and I need to know the return code set by VSAM when a function like `kretrv` or `kinsert` fails. How can I find this information?

A. When the SAS/C library invokes an operating system routine, such as a VSAM macro, and the macro fails, information about the failure is saved in a system macro information structure. You can access the name of the macro that most recently failed via the library macro `__sysmi_macname` and its return code via `__sysmi_rc`. For more information on this facility, see "System Macro Information" on page 1-8 .

Sharing an Output PDS

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 MVS 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 your application, if you are certain that only one user can open the file for output at a time, you should access the file through a DDname rather than through a data set name. You can define the DDname using JCL or a TSO ALLOCATE command as SHR, and the library will not alter this allocation when the DDname is opened. In TSO, you can use the `system` function to allocate a data set to a specific DDname. Also, in any environment, you can use the `osdynalloc` function to dynamically allocate the data set.

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.

4 Environment Variables

The Environment Variable Concept

The ISO/ANSI C standard requires that a C library include the **getenv** function, which retrieves the value of a named environment variable. Both the name and value are character strings. The ISO/ANSI C standard does not describe the meaning of environment variables, how they might be set, or even whether it is possible to define them. However, the **getenv** function should be similar to the **getenv** function under the UNIX environment, and if the host operating system has a facility similar to UNIX environment variables, **getenv** should provide a means of accessing that facility.

These properties are fundamental to the behavior of UNIX environment variables:

- Environment variables can be set and tested by shell commands. For example,

```
export HOME=/u/george
```

- A program can modify its own environment by adding or replacing variables; however, these changes are local to the program.
- When one program calls another (for example, using the **system** function), the calling program's environment variables are passed to the called program. The communication is one way; that is, the caller passes the environment to the called program, but any changes the called program makes to its environment are not available to the caller.
- Special forms of the **exec** function enable a program to pass control to another program with a specially built environment.
- Because all running programs have a shell as an ancestor, the environment maintained by the shell is inherited by all of a session's environments, subject to modification by specific programs.

Secondary properties of UNIX environment variables include

- Both environment variable names and environment variable values are case sensitive and have no practical length limits.
- The shell only permits environment variable names containing alphanumeric characters and underscores. (However, C programs can have defined variables not described by this rule.) By convention, environment variables meaningful to the system are entirely in uppercase, leaving lowercase names for individual applications.
- Each program has an external variable, **`environ`**, which addresses the list of environment variables. Many older programs access **`environ`** for environmental information rather than calling **`getenv`**.
- The only portable way of modifying the environment is by using **`environ`**. Some versions of the UNIX environment implement functions like **`putenv`** to modify the environment, but these functions are not portable.

The most difficult aspect of implementing environment variables under the MVS and CMS operating systems is that native methods for invoking other programs (ATTACH, CMSCALL) do not provide for a list of variables. For this reason, the SAS/C environment variable implementations provide somewhat different semantics in which variables are shared by all programs in a session, regardless of how the programs are invoked. The implementation was modeled after CMS global variables (also called GLOBALV variables) so that C environment variables could be used as a way of accessing and modifying these variables under CMS. This implementation is adequate for porting many UNIX programs that use environment variables, in particular, for all programs that do not modify their own environment. SAS/C software implements environment variables similar to GLOBALV under CMS, TSO, and CICS. Also, for programs running under the OpenEdition shell, the environment variable implementation complies with POSIX.1 and traditional UNIX usage.

CMS Global Variables

CMS global variables are similar to C environment variables but have many differences in the details. Some of the differences are

- Global variables can be manipulated from the CMS command line using the GLOBALV command.
- Global variables are shared between all programs in a session. Changes made by one program are visible to all other programs.
- There are several forms of global variables: storage, session, and lasting. Storage variables retain their values only until the next IPL command of CMS; session variables last until the end of a session; lasting variables keep their values across CMS sessions.
- Global variables are initialized from files with known names and formats. By editing the files, you can change the initial values used for global variables the next time you invoke CMS.
- Global variables are organized into groups of related variables with a default group named UNNAMED.
- Environment variable names are translated to uppercase letters automatically. Both names and values are limited to 255 characters.
- CMS does not provide an interface for a program to determine the names of all defined global variables.

SAS/C Environment Variables

SAS/C combines elements of UNIX environment variables and CMS global variables in its own environment variable implementation. The interface has been made as portable as is reasonable, but there is still a large amount of system dependency in this implementation. The most important extensions are environment variable scopes and environment variable groups.

Environment Variable Scopes

SAS/C defines three scopes of environment variables: program, external, and permanent.

Program-scope environment variables
: are most similar to UNIX environment variables. They are strictly local to a program; thus, changes to a program-scope variable are not visible to any other program. Program-scope environment variables are not passed to invoked programs unless one of the OpenEdition **`exec`** functions is used.

  Program-scope environment variables are set in one of two ways. First, they may be specified on the program's command line as described in Chapter 8, "Run-Time Argument Processing," in the . Or, for programs invoked by the OpenEdition **`exec`** system call, they may be passed by the caller of **`exec`**.

  There are no limits on the size of program scope variable names or values. The names are case insensitive; that is, **`home`**, **`Home`**, and **`HOME`** are considered to identify the same variable. (See "OpenEdition Considerations" on page 4-4 for an exception.)

External-Scope Environment Variables
: correspond to CMS storage GLOBALV variables. They can be set outside of a program using a system command such as GLOBALV for CMS or PUTENV for TSO. Any changes made by a program are visible to other programs, but all such changes are lost at the end of a session. External-scope environment variables are case insensitive and may have length limitations imposed by the host system.

Permanent-Scope Environment Variables
: correspond to CMS lasting GLOBALV variables. Their behavior is the same as external-scope variables except that changes to these variables persist to future sessions.

SAS/C implements program-scope environment variables in all environments. It implements external- and permanent-scope environment variables under TSO, CMS, and CICS.

When a program calls the **`getenv`** function to retrieve the value of an environment variable, the scopes are searched in order. That is, first the program-scope variables are checked, then the external-scope variables, and then the permanent-scope variables. This enables you or the program to override the usual value with a more temporary value for the duration of a program or session.

When a program calls the **`putenv`** or **`setenv`** function to modify or add an environment variable, the change is made at program scope unless some other scope is explicitly specified. Thus, a portable program that changes its own environment will not affect any other program. Scopes are indicated to **`putenv`** by a prefix on the environment variable name followed by a colon, for example, **`external:TZ`**. Because colons are not commonly used in UNIX environment variable names, this extension has little effect on portability.

Environment Variable Groups

Another extension in the SAS/C environment variable implementation supports group names for external- and permanent-scope variables. If an external- or permanent-scope environment variable name contains a period, the portion of the name before the period is considered to be the group name. For example, the variable name **LC370.MACLIBS** defines the variable **MACLIBS** in the group **LC370**. An environment variable without a group name is considered to have a default group name of **CENV**.

CMS group names are limited to eight characters. For this reason, in certain cases, environment variable names containing periods may behave differently in different scopes. For example, the two environment variables **TOOLONGFORME.NAME** and **TOOLONGFORANYONE.NAME** refer to the same variable for an external scope but different variables for a program scope. This should not be a problem in normal usage.

Because environment variables in traditional UNIX usage do not normally contain periods, this extension does not ordinarily cause portability problems.

OpenEdition Considerations

The POSIX standards implemented by OpenEdition make certain requirements on the C library that are not compatible with the SAS/C environment variable implementation defined in the previous section. These requirements are

- POSIX.1 requires that environment variable names be case sensitive. For compatibility with CMS, SAS/C treats environment variable names as case insensitive. This problem is handled in two ways:
  - For programs called by the shell (or more generally by any **exec** function), program-scope environment variables are considered case sensitive. For all other programs, these names remain case insensitive for compatibility with CMS and previous releases.
  - For non-**exec**-linkage programs, the original form of the variable is preserved even though the case of an environment variable is not considered significant. (That is, the variable **MyName** is stored as **MyName**, not as **myname** or **MYNAME** even though these are all names for the same variable.) If a program calls **exec** to pass control to an **exec**-linkage program, the called program receives the environment variables with their original case information intact.
- POSIX.1 requires that all environment variables be accessible by the external variable **environ**. SAS/C now makes the program-scope environment variables accessible by **environ** for all programs. However, variables of other scopes are never accessible by **environ**. Similarly, when an **exec** function is called, only the program-scope environment variables are passed to the called program.
- POSIX.1 requires that it be possible to modify the environment variable list by manipulating **environ** and the list it addresses. SAS/C supports this modification but does not recommend it. The POSIX.1a function **setenv** should be used to update the environment whenever possible. In particular, you should not use both **setenv** and update **environ** simultaneously. This is a restriction stated by the POSIX.1a draft standard.
- The POSIX use of the **environ** variable is not strictly ISO/ANSI conforming because **environ** is in the name space reserved by ISO/ANSI for users. For this reason, the name **environ** is accessible only to programs compiled with the **posix** option. Additionally, the name **environ** is valid only in the **main** load module of a program. To access **environ** from another load module, declare

  ```
  extern char ***_environ;
  ```

  and use ***_environ** in place of **environ**. This produces correct results in any load module but is not portable. **`_environ`** can be used in programs that are not compiled with the **`posix`** option.

- SAS/C supports the POSIX.1a **`clearenv`** function, which removes all environment variables. It has an effect only on program-scope variables. That is, external- and permanent-scope variables are unaffected.

Environment Variables under CMS

External- and permanent-scope environment variables are implemented as CMS global and lasting variables under CMS. CMS limits the names and values of these variables to 255 characters. See the *CMS Command Reference* for more information on GLOBALV.

Environment Variables under TSO

Under TSO, environment variables are implemented as a SAS/C extension. SAS/C provides GETENV and PUTENV TSO commands enabling you to inspect and modify environment variables without writing a C program to do so. These commands are described in Chapter 6, “Executing C Programs,” in the *SAS/C Compiler and Library User’s Guide.*

Note: Under CMS, when you update a permanent-scope environment variable, the new value is also assigned to the external-scope variable of the same name.

Environment variable names are limited under TSO to 254 characters, and values are limited to 255 characters.

TSO Technical Notes for Environment Variables

This section describes technical aspects of the SAS/C TSO environment variable implementation.

Accessing the Environment Variable File

The following strategy is used to locate the PERMANENT environment variable file:

1. If the DDname C@ENV is defined, the file allocated to that DDname is used. If this fails and TSO is running in batch, no attempt is made to allocate a file and no PERMANENT environment variables are assumed to exist.
2. If the DDname C@ENV is not defined, the data set *userid*.C@ENV.PERM is allocated to the DDname. Note that the first data set qualifier is always the userid even when it differs from the user’s default TSO prefix.
3. If the data set *userid*.C@ENV.PERM does not exist, it is created if the request is a **`putenv`** function with a PERMANENT scope. If the request is a **`getenv`** function, the file is not created.

Environment Variable File Format

Each line of *userid*.C@ENV.PERM either defines an environment variable or a group. The variables of each group are defined after the group definition. Variables defined before the first group are part of the default group **`CENV`**. A line of the form =*group* defines a group, and a line of the form *var*=*value* defines a variable. If *var*=*value* uses more than 254 characters, it takes up two lines in the file and is split at the equal sign.

Assume that the content of a *userid*.C@ENV.PERM file is as follows:

NAME=*Fred*
MISC.AVERYLONGNAME=*averylongvalue*

In this example, two PERMANENT environment variables are defined: NAME has a value of *Fred*, and MISC.AVERYLONGNAME has a value of *averylongvalue.*

Environment Variable Implementation

This information is provided for TSO systems programmers. Most users will not need this information.

TSO environment variables are kept in subpool 78 memory for the life of the session. They are located through an anchor field at offset 260 from the RLGB (relogon buffer), which is a field reserved by IBM. Before using this field, the C environment variable routines check the field to see if it is already being used by either IBM or the site. If the field is unavailable, then EXTERNAL environment variables are also stored in a file, rather than in memory. In this case, access is slower because the variables are not in memory.

If the DDname T@ENV is allocated, the DDname is used for EXTERNAL environment variables. If T@ENV is not allocated, a temporary data set is created and allocated to this DDname.

The TSO environment variable implementation stores environment variables in shared subpool 78 storage. If a task that does not share subpool 78 with the rest of TSO invokes **`putenv`**, the **`putenv`** function fails. Also, use of **`setenv`** by such tasks may involve substantial extra processing.

Environment Variables under MVS Batch

Only program-scope environment variables are supported under MVS batch, except when you run the TSO terminal monitor program. The argument redirection facility described in Chapter 8, “Run-Time Argument Processing,” in the *SAS/C Compiler and Library User’s Guide* enables you to define as many environment variables as you require without concern for the 100-character limit on the JCL PARM string.

Environment Variables under CICS

Environment Variable Implementation

Nonprogram-scope environment variables are stored in a temporary storage queue, which exists until it is manually deleted or the CICS region (or temporary storage) is cold started. Thus, the lifetime of a queue can vary considerably. Some sites cold start their regions every night. If CICS crashes during the day, it could also be cold started at that time. Generally, environment variables persist for the duration of a session connection (logon to logoff), but they could persist between connections.

By default, the library uses the VTAM netname as a queue name and only accesses the first record in the queue. The queue name is set in the user-replaceable routine L$UEVQN. This routine enables a site to use some other scheme for assigning a queue name, including the option of sharing a single queue between all programs. See the source code in the SASC.SOURCE data set (under MVS) or the LSU MACLIB (under CMS) for complete details.

Because Temporary Storage is a shared facility on CICS, some potential problems exist with the technique of using the VTAM netname as a queue name. Some of those problems are:

- VTAM netnames can be reused on some systems so that one user could wind up with another user’s environment variables.
- Someone other than you might logon to the terminal where you were sitting; they would then get your environment variables.
- A user logging on might access the environment variables defined by the previous user of the terminal.

Scope and Size of CICS Environment Variables

Only program- and external-scope environment variables are supported under CICS (that is, there is no support for permanent or lasting environment variables). Note that for CICS, session scope is considered to be the same as external or storage scope.

Environment variable names are limited in CICS to 254 characters, and values are limited to 255 characters.

SAS/C Supplied Transactions to Inspect and Modify Environment Variables

SAS/C provides a transaction that enables you to inspect and modify environment variables without writing a program to do so. You invoke the transaction to set the environment variables of interest; then, in a separate step, you invoke your application transaction. See the *SAS/C CICS User's Guide, Second Edition* for more information.

5 Signal-Handling Functions

Introduction

The signal-handling feature of the SAS/C library is a collection of library functions that enables you to handle unexpected conditions and interrupts during execution of a C program. Using this facility, you can

- define a function, called a signal handler, that performs special processing when a signal occurs
- control which signals are processed by your program, which are ignored, and which are handled using the system default action
- block the reception of some signals
- generate signals
- define signals for your own needs.

Release 6.00 of the SAS/C library provides OpenEdition and POSIX support for signal handling. Consequently, many new signals and signal-handling functions have been added to the library. Wherever possible, the SAS/C library preserves the behavior of previously existing signal-handling programs and treats POSIX signals and non-POSIX signals similarly. Where the POSIX definition of a signal and the SAS/C definition differ, a choice of behavior is offered on a per-signal basis.

Terminology Used in Signal Handling

This section introduces some of the terminology used to discuss the signal-handling features of the SAS/C library.

interrupt	is a hardware or software event that causes current processing to be suspended while the condition causing the interruption is processed. An interrupt can be processed either by the operating system or an application program or both.
signal	is an interrupt processed by the library, usually after processing by the operating system.
synchronous signal	is a signal directly resulting from program execution. For example, floating-point division by 0 generates a synchronous signal.
asynchronous signal	is a signal resulting from an interrupt external to the program. The timing of an asynchronous interrupt, in terms of program execution, is not predictable. For example, if you press the ATTN key while running a program under TSO, an asynchronous signal is generated. Because any program can send an OpenEdition signal to any other program, any signal defined by OpenEdition can be asynchronous.
discovery of a signal	is the time at which the library suspends normal program execution to respond to a signal. Synchronous signals are always discovered immediately, but asynchronous signals can only be discovered at certain points in execution, as described in the next section, "Types of Signal Support." Discovery of an asynchronous signal does not have to take place immediately after the occurrence of the original interrupt.
pending	is the state of a signal between the time of its occurrence and the time it is discovered.
signal blocking	is a processing technique for postponing discovery of an asynchronous signal. The program may specify one or more asynchronous signals that are to be blocked, and the set of blocked signals may be changed at any time. When a blocked signal occurs, program execution is unaffected until the program unblocks the signal. After it is unblocked, the signal will be discovered.

Types of Signal Support

With Release 6.00, signals are supported either directly by the SAS/C library or in cooperation with OpenEdition. To provide backward compatibility, a few differences exist between the way the SAS/C library manages signals and the way OpenEdition, with its POSIX support, manages them.

SAS/C Library

The SAS/C library distinguishes between synchronous and asynchronous signals. You can define signal handlers for both types of signals, but the timing for executing a handler differs depending on the type of signal. When a synchronous signal occurs, the handler for that signal is called immediately. When an asynchronous signal occurs, the library delays calling the handler until a function is called or returns or until the special function **`sigchk`** is called to discover pending asynchronous signals. See "Discovering Asynchronous Signals" on page 5-12 for details on how the SAS/C library treats asynchronous signals.

Synchronous Signals

The synchronous signals defined by the library are

`SIGABND`	`SIGFPE`	`SIGIDIV`	`SIGSEGV`
`SIGABRT`	`SIGFPOFL`	`SIGILL`	`SIGTERM`
`SIGFPDIV`	`SIGFPUFL`	`SIGMEM`	

Eight synchronous user signals (`SIGUSR1 through SIGUSR8`) are also available.

Asynchronous Signals

The asynchronous signals defined by the library are

`SIGALRM`	`SIGINT`	`SIGIUCV`

Eight asynchronous user signals (`SIGASY1 through SIGASY8`) are also available.

OpenEdition and POSIX

The POSIX.1 standard defines a large number of signals that may be sent to a process. Signals may be sent synchronously or asynchronously, but this does not depend on signal type. For instance, the signal **`SIGPIPE`** may be raised synchronously by OpenEdition when an attempt is made to read a pipe without a writer process, or it may be raised asynchronously by another process's use of the **`kill`** function to send this signal. Each signal has a default action, which is usually abnormal process termination, with some exceptions. For example, the default handling for the **`SIGTSTP`** signal is to suspend process execution.

Almost all signals can be blocked, which means they are delayed from having an effect on the target process until that process unblocks the signal. (Two signals, **`SIGKILL`** and **`SIGSTOP`**, cannot be blocked. These signals also cannot be handled and, therefore, always cause their default actions.)

Note: This definition differs considerably from the way SAS/C previously implemented signals. Traditionally, SAS/C only allowed asynchronous signals to be blocked. The synchronous/asynchronous distinction does not apply to POSIX signals, and blocking has an effect even on synchronously generated signals like **`SIGPIPE`**, with a few exceptions.

Supported Signals

The signals supported by Release 6.00 can be divided into three groups: signals managed only by SAS/C, signals managed exclusively by OpenEdition, and signals with dual support. For dual-support signals, you can define during program startup which signals are to be handled by OpenEdition and which are to be handled by the SAS/C library.

Signals managed exclusively by SAS/C are

`SIGASY1-8`	application-defined asynchronous signals
`SIGFPDIV`	floating-point division by 0
`SIGFPOFL`	floating-point overflow
`SIGFPUFL`	floating-point underflow
`SIGIDIV`	integer division by 0

`SIGIUCV`	VM inter-user communication signal
`SIGMEM`	no memory available for stack space
`SIGUSR3-8`	application-defined synchronous signals.

Signals managed exclusively by OpenEdition are

`SIGCHLD`	child process stopped
`SIGCONT`	continue process if stopped
`SIGHUP`	controlling terminal hangup
`SIGKILL`	process terminated (cannot be blocked)
`SIGPIPE`	write to a pipe that is not open for reading
`SIGQUIT`	interactively terminate process
`SIGSTOP`	process stopped (cannot be blocked)
`SIGTRAP`	debugging signal
`SIGTSTP`	interactively stop process
`SIGTTIN`	background process read from controlling terminal
`SIGTTOU`	background process write to controlling terminal.

Signals managed either by SAS/C or OpenEdition are

`SIGABND`	system ABEND
`SIGABRT`	abort function called
`SIGALRM`	real-time expiration signal
`SIGFPE`	general computational error
`SIGILL`	illegal instruction executed
`SIGINT`	interactive terminal attention signal
`SIGIO`	undefined
`SIGSEGV`	illegal memory access
`SIGTERM`	termination request
`SIGUSR1-2`	application-defined synchronous signals.

Each signal that may be managed by SAS/C is described in detail later in this section. See the POSIX.1 standard or *The POSIX.1 Standard: A Programmer's Guide* (Zlotnick 1991), for information on the signals defined by POSIX and managed by OpenEdition. All signals that can be used with the library have assigned numbers in the header file **`<signal.h>`**. Always refer to signals by their names, not their numbers, because signal numbers are not portable.

Note: The library permits you to raise the signals **`SIGIO`** and **`SIGTERM`**, but at the present it does not assign a meaning to these signals.

Choosing Signal Support Using oesigsetup

The **`oesigsetup`** function enables you to control which signals are managed by OpenEdition and which use support internal to the SAS/C library. **`oesigsetup`** must be called before any other signal-related function.

If there is no call to **`oesigsetup`** in a program called with **`exec`** linkage, the library assumes that all signals should be managed by OpenEdition, if possible. If there is no call to **`oesigsetup`** in a program not called with **`exec`** linkage (a regular batch or TSO execution), the library assumes that no signals should be managed by OpenEdition.

Note: This means that you *must* call **oesigsetup** in a program without **exec** linkage if you need to use OpenEdition signals.

The arguments to **oesigsetup** are two signal sets. The first defines the set of signals to be managed by OpenEdition, and the second defines the signals to be managed by SAS/C. **oesigsetup** fails if any signal is included in both sets; unspecified signals are handled as if **oesigsetup** had not been called.

Error Signals with Flexible Handling

The signals with flexible handling can be divided into two groups: error signals and other signals. The error signals, normally associated with program error condition, are **SIGABND**, **SIGABRT**, **SIGFPE, SIGILL**, and **SIGSEGV**. If these signals are handled as SAS/C conditions, OpenEdition is not informed of the error when the corresponding error condition occurs. If the error leads to termination, OpenEdition sets the final status of the terminated process to *terminated by* **SIGKILL** rather than a more specific status; otherwise, you can expect no undesirable effects.

Note: The **SIGABND** signal is used in Release 6.00 of SAS/C software to indicate a system ABEND. As described in the next section, this is a behavior change from previous versions of SAS/C.

Non-Error Signals with Flexible Handling

The non-error signals with flexible handling are **SIGALRM**, **SIGINT**, **SIGIO**, **SIGTERM**, **SIGUSR1**, and **SIGUSR2**. If OpenEdition handles **SIGALRM**, the SAS/C extension functions **alarmd** and **sleepd** are not available. If the SAS/C library handles **SIGALRM**, the **ps** shell command does not accurately indicate when the process is sleeping.

If SAS/C handles **SIGINT**, **SIGINT** is generated by use of the TSO attention key for a program running under TSO. If OpenEdition handles **SIGINT**, SAS/C does not use the **STAX** macro or attempt to handle TSO attentions. Be aware that SAS/C handling of **SIGINT** is not useful in non-TSO address spaces.

SIGIO has no special meaning at present to either OpenEdition or SAS/C but might be used by future versions of either product.

SIGTERM has no defined meaning to SAS/C and, therefore, can be generated only by use of the **raise** function if managed by SAS/C.

SIGUSR1 and **SIGUSR2** have no special meaning to OpenEdition. If **oesigsetup** defines these signals as managed by SAS/C, then you can use SAS/C user-added signal support to define a meaning for one of these symbols.

Note: If you have defined a signal as handled by SAS/C and the signal is generated by OpenEdition, the result is always the OpenEdition default action for the signal. For example, if you define **SIGTERM** as a SAS/C signal and establish a handler, after which another process uses the **kill** function to send your process a **SIGTERM** signal, your handler will not be called and the process will be terminated.

A program can use **kill** to send a signal that **oesigsetup** has defined as a signal managed by SAS/C. If a program sends the signal to itself, only default handling will take place.

Handling Signals

Either the SAS/C library or OpenEdition defines a default action for each signal that can occur. If you want to override the default action for a signal, you can define a function called a signal handler that performs your own signal-handling actions. You can also request that the library or OpenEdition ignore some signals.

A signal handler may be identified by either the **signal** or the **sigaction** function, regardless of whether the signal is managed by SAS/C or OpenEdition. In general, **sigaction** is more flexible because it does not have to be reissued every time you enter a handler. However, **signal** is more portable because it is defined by the ISO/ANSI standard and it is defined by traditional UNIX C compilers; whereas, the

sigaction function is defined by the POSIX standard and is often not available with traditional UNIX C compilers.

Using signal

These basic actions are used to define a signal handler using the **signal** function defined by the ISO/ANSI standard:

- Call the **signal** or **sigaction** functions to identify a handler for the signal in the beginning of the program or at some point before the signal may occur.
- Code the handler function.
- Optionally, call **signal** within the handler function to reinstate handling for that signal. (This is not necessary when a handler is defined using **sigaction**).

The **signal** function requires two arguments: the name of the signal and the address of the function that handles the signal when it occurs. When a signal occurs, the handler defined by use of the **signal** function is called. The handler can do anything a normal C function can do. Frequently, a handler calls **exit** or **abort** to terminate program execution, or it calls **longjmp** to transfer control to a higher-level function. (For example, a handler for **SIGINT** may use **longjmp** to transfer control to the main function to immediately get new input from the terminal.)

After processing a signal, you may want the program to continue execution from the point at which it was interrupted. In this case, the handler simply executes a **return** statement. However, for some signals (such as **SIGILL**), it is impossible to resume at the point at which the signal occurred. If a signal handler returns in such a situation, the program is abnormally terminated.

The call to **signal** establishes signal handling for only *one* occurrence of a signal. Before the signal-handling function is called, the library resets the signal so that the default action is performed if the same signal occurs again. Resetting signal handling helps to prevent an infinite loop if, for example, an action performed in the signal handler raises the same signal again. If you want your handler to be used for a signal each time it occurs, you must call **signal** within the handler to reinstate it. You should be cautious in reinstating signal handling. For example, if you continually reinstate **SIGINT** handling, you may lose the ability to interrupt and terminate your program.

Using sigaction

These basic actions are used to define a signal handler using the **sigaction** function defined by the POSIX standard:

- Use the **sigemptyset**, **sigfillset**, **sigaddset**, **sigdelset**, and **sigismember** functions as required to define a signal mask that will block signals while the handler is running.
- Optionally, set flags that modify the behavior of the signal being handled.
- Call the **sigaction** function to identify a handler for the signal at the beginning of the program or at some point before the signal may occur.
- Code the handler function.

As you can see from the previous list, the signal handling process used with **sigaction** is significantly different from the one used with **signal**. The **sigaction** structure, which contains information that describes the action to be performed when a signal is received, is defined as:

```
struct sigaction {
   _HANDLER sa_handler;
   sigset_t sa_mask;
   int sa_flags;
};
```

sa_handler is a pointer to the signal handler; **sa_mask** is the signal mask that will be used while the signal handler is running, and **sa_flags** specifies the flags that affect the handling of the signal.

Here is a brief example of how you can use the **sigaction** structure to define the action that should be performed when a **SIGQUIT** signal is received:

```
extern void quitHandler(int);

#include <unistd.h>
#include <signal.h>

main()
{
   struct sigaction quitAction;

   quitAction.sa_handler = quitHandler;
   sigfillset(&quitAction.sa_mask);
   quitAction.sa_flags = 0;
   sigaction(SIGQUIT, &quitAction, NULL);
   .
   .
   .
}

void quitHandler(int signum)
{
   .
   .  /* Signal handler code goes here. */
   .
}
```

If this code is executed, the **quitAction** signal handler is called whenever a **SIGQUIT** signal is received. The call to **sigfillset** causes the **quitAction.sa_mask** to block all signals during the execution of the signal handler. Any signals blocked by the signal-action mask are delivered after the handler returns. Setting the **quitAction.sa_flags** to 0 specifies that no flags are used to modify the behavior of the signal. See the description of **sigaction** for a discussion of the signal action flags that you can specify.

Notice that the **sigaction** function does not require you to reinstate the signal handler within the signal handling function itself. The handling of a signal identified by **sigaction** is *not* reset to the default action after an occurrence of the signal. This behavior is different from the behavior of a signal handler identified by the **signal** function.

ABEND Handling

Before Release 6.00, SAS/C used the signal **SIGTERM** to represent a system ABEND of a C program. OpenEdition defines the signal **SIGABND** for this purpose. Therefore, with Release 6.00, SAS/C uses **SIGABND** rather than **SIGTERM** to represent an ABEND to be compatible with OpenEdition and to allow programs to handle **SIGTERM** in a more portable fashion.

Note: **SIGABND** has been assigned the old signal number for **SIGTERM** and **SIGTERM** has been given a new signal number. Thus, existing load modules that use the **SIGTERM** signal to intercept system ABENDs continue to work until they are recompiled.

SAS/C uses the signal **SIGABRT** to represent user ABENDs including library ABENDs. This is different from OpenEdition, which expects all ABENDs to be treated as **SIGABND**.

Note: If library ABEND handling is suppressed by the **=nohtsig** run-time option, or if library errors occur handling an ABEND, OpenEdition reports the status of an ABEND as *terminated by* **SIGKILL** rather than as *terminated by* **SIGABND**.

Getting More Information about a Signal

For some signals, you can get more information about the signal by calling the **siginfo** function in the signal handler. The **siginfo** function returns a pointer to the information associated with the signal being handled.

For example, when **SIGMEM** is raised, a call to **siginfo** provides information on the number of bytes of memory needed for additional stack space. If **SIGFPE** is raised, **siginfo** returns pointers to data that can be modified by the handler. For example, when **SIGFPE** is raised by a floating-point overflow, the information returned by **siginfo** includes the result of the computation that caused the overflow. You can replace this value and resume execution.

Some signals have no additional information available. If you call **siginfo** for one of these signals or if no signal has occurred, **siginfo** returns **NULL**. Refer to the descriptions of each signal for details on the information returned by **siginfo**. Table 5.1 summarizes the information returned by **siginfo**.

Note: **siginfo** is not a standard C function; avoid using it in programs that must be portable.

Table 5.1
Summary of Information from siginfo

Signal	Information Returned by siginfo for Signals Raised Naturally
SIGABND	pointer to structure of type ABND_t
SIGABRT	pointer to structure of type ABRT_t
SIGALRM	NULL
SIGFPE	pointer to structure of type FPE_t
SIGFPDIV	pointer to structure of type FPE_t
SIGFPOFL	pointer to structure of type FPE_t
SIGFPUFL	pointer to structure of type FPE_t
SIGIDIV	pointer to structure of type FPE_t
SIGILL	pointer to structure of type ILL_t
SIGINT	NULL
SIGIUCV	pointer to various types of structures; refer to Chapter 5, "Inter-User Communications Vehicle (IUCV) Functions," in the *SAS/C Library Reference, Third Edition, Volume 2, Release 6.00.*
SIGMEM	pointer to integer
SIGSEGV	pointer to structure of type SEGV_t
SIGTERM	NULL

The return value of **siginfo** is always 0 for any signal managed by OpenEdition, unless the signal was generated by **siggen** or the signal was a program check or ABEND directly associated with a program error.

Default Signal Handling

If you do not define a special handler for a signal or invoke a signal handler with the `signal` function and do not reinstate signal handling, the library performs default actions specific to each signal. You can also invoke default signal handling by using the special action `SIG_DFL` as the second argument to `signal`. For most signals, the default action is to abnormally terminate the program. Detailed discussions of default actions are in "Signal Descriptions" on page 5-20 ; default actions are listed in Table 5.2.

Table 5.2 *Summary of Default Actions*

Signal	SAS/C Library Default Action (`SIG_DFL` Handler)	OpenEdition Default Action (`SIG_DFL` Handler)
`SIGABND`	ABEND with system code ABE	ends the process
`SIGABRT`	ABEND with user code 1210	ends the process
`SIGALRM`	ABEND with user code 1225	ends the process
`SIGCHLD`	not supported	signal is ignored
`SIGCONT`	not supported	continues a stopped process; otherwise, the signal is ignored
`SIGFPDIV`	ABEND with 0CF.	not supported
`SIGFPE`	raises another signal: `SIGFPOFL`, `SIGFPUFL`, `SIGFPDIV`, or `SIGIDIV`. Refer to the descriptions of these signals for defaults.	ends the process
`SIGFPOFL`	ABEND with 0CC	not supported
`SIGFPUFL`	changes result of computation to 0; execution continues	not supported
`SIGHUP`	not supported	ends the process
`SIGKILL`	not supported	ends the process
`SIGIDIV`	ABEND with 0C9	not supported
`SIGILL`	ABEND with appropriate code (0C1, 0C2, 0C3, or 0C6)	ends the process
`SIGINT`	no default actions by library; TSO default action is to ABEND; CMS default action ignores the signal	ends the process
`SIGIO`	ignored	ignored
`SIGIUCV`	ABEND with user code 1225	not supported
`SIGMEM`	attempts to continue execution; ABEND with code 80A (under TSO) or 0F7 (under CMS) if more than 4K of stack space is required.	not supported
`SIGPIPE`	not supported	ends the process

continued

Table 5.2 (*continued*)

Signal	SAS/C Library Default Action (`SIG_DFL` Handler)	OpenEdition Default Action (`SIG_DFL` Handler)
`SIGQUIT`	not supported	ends the process
`SIGSEGV`	ABEND with appropriate code (0C4 or 0C5)	ends the process
`SIGSTOP`	not supported	stops the process
`SIGTERM`	ABEND with user code 1225	ends the process
`SIGTRAP`	not supported	ends the process
`SIGTSTP`	not supported	stops the process
`SIGTTIN`	not supported	stops the process
`SIGTTOU`	not supported	stops the process

Ignoring Signals

If you want to ignore the occurrence of a signal, specify the special action **`SIG_IGN`** as the second argument to **`signal`**. Ignoring a signal causes the program to resume execution at the point at which the signal occurred. Some signals, such as **`SIGABRT`**, cannot be ignored because it is impossible to resume program execution at the point at which these signals occur.

Also, some signals such as **`SIGSEGV`** should not be ignored when they are managed by OpenEdition, even though it is possible to do so because the results are unpredictable. These signals are ignored if the specified action is **`SIG_IGN`** and they are generated by a call to the **`kill`** function; however, if the signal is generated by either a program check or an ABEND, the most likely result is that another ABEND will occur.

Table 5.3 lists which signals can be ignored. For more information, refer to the descriptions of the signals.

Table 5.3 *Summary of Ignoring Signals*

Signal	SAS/C Library Ignored Signals (`SIG_IGN` Handler)	OpenEdition Ignored Signals (`SIG_IGN` Handler)
`SIGABND`	cannot be ignored; ABEND as described in Table 5.2	should not be ignored; results unpredictable
`SIGABRT`	cannot be ignored; ABEND as described in Table 5.2	should not be ignored; results unpredictable
`SIGALRM`	program continues; signal has no effect	program continues; signal has no effect
`SIGCHLD`	not supported	program continues; signal has no effect
`SIGCONT`	not supported	program continues; signal has no effect
`SIGFPDIV`	program continues; result of computation undefined	not supported

continued

Table 5.3 (*continued*)

Signal	SAS/C Library Ignored Signals (`SIG_IGN` Handler)	OpenEdition Ignored Signals (`SIG_IGN` Handler)
`SIGFPE`	program continues; result of computation undefined except for **`SIGFPUFL`** signals (see **`SIGFPUFL`**)	same as SAS/C
`SIGFPOFL`	program continues; result of computation undefined	not supported
`SIGFPUFL`	program continues; result of computation set to 0	not supported
`SIGHUP`	not supported	program continues; signal has no effect
`SIGKILL`	not supported	cannot be ignored
`SIGIDIV`	program continues; result of computation undefined	not supported
`SIGILL`	cannot be ignored; ABEND as described in Table 5.2	should not be ignored; results unpredictable
`SIGINT`	use of ATTN or PA1 key under MVS, or IC command under CMS; signal has no effect	program continues; signal has no effect
`SIGIO`	program continues; signal has no effect	program continues; signal has no effect
`SIGIUCV`	cannot be ignored; ABEND as described in Table 5.2	not supported
`SIGMEM`	execution continues until storage exhausted	not supported
`SIGPIPE`	not supported	program continues; signal has no effect
`SIGQUIT`	not supported	program continues; signal has no effect
`SIGSEGV`	cannot be ignored; ABEND as described in Table 5.2	should not be ignored; results unpredictable
`SIGSTOP`	not supported	cannot be ignored
`SIGTERM`	program continues; signal has no effect	not supported
`SIGTRAP`	not supported	program continues; signal has no effect
`SIGTSTP`	not supported	program continues; signal has no effect
`SIGTTIN`	not supported	program continues; signal has no effect
`SIGTTOU`	not supported	program continues; signal has no effect

Generating Signals

In the normal execution of a program, signals occur at unpredictable times. As you write and test a program that handles signals, you may want to generate signals to ensure that your program handles them correctly. Your program also may need to generate signals as part of an error-checking routine. For example, a mathematical function may generate a **`SIGFPOFL`** signal if it determines that an overflow is certain to occur during its processing.

The library provides three functions for generating signals: **`raise`**, **`siggen`**, and **`kill`**.

- **`raise`** is an ISO/ANSI Standard function that raises the signal you pass as the argument.
- **`siggen`** is provided by the SAS/C library; it is not portable. Besides raising the signal you pass, **`siggen`** also enables you to define the value returned by a subsequent call to **`siginfo`**.
- **`kill`** is a POSIX function used to send a signal to a process. A process can use the **`kill`** function to send a signal to itself. However, **`kill`** does not support SAS/C signals, and it can only be used with signals managed by OpenEdition.

If you raise a signal with **`raise`** or **`siggen`** the handler is called immediately, even if the signal is asynchronous, unless it is a blocked signal managed by OpenEdition. Therefore, these functions are not useful for testing signal blocking.

If you use **`raise`** or **`siggen`** to generate a signal with no special handler defined and the default action is abnormal termination, the program abnormally terminates. However, this abnormal termination may not be exactly the same as it would be if the signal had occurred naturally.

User-defined signals (**`SIGUSR1 through SIGUSR8`** and **`SIGASY1 through SIGASY8)`** can be generated by using **`raise`** or **`siggen`**. Refer to Chapter 12, "User-Added Signals," in the *SAS/C Library Reference, Volume 2* for another method of raising user-defined signals.

Discovering Asynchronous Signals

As mentioned earlier, the library calls handlers for synchronous signals as soon as the signal occurs; however, when an asynchronous signal occurs, the library may not immediately call the handler. There may be a delay because the signal must be discovered by the run-time library. After the signal is discovered, the library calls the handler. There are only three times asynchronous signals are discovered:

- when a function is called
- when a function returns
- when the special function **`sigchk`** is called to discover pending asynchronous signals.

You can insert calls to **`sigchk`** in your program to decrease the number of statements that are executed before a signal is discovered.

The library limits the times that an asynchronous signal can be discovered to improve your control of signal handling. There are two reasons the SAS/C library delays processing asynchronous signals:

- If the library allowed the operating system to call signal handlers as soon as a signal occurred, the signal handler would not be able to use facilities such as **`exit`** and **`longjmp`**. Using these facilities bypasses return to the operating system, which causes unpredictable results. By limiting signal discovery (and signal handling) to times when the library has control, the library permits you to use all of the facilities of C, including **longjmp** and **exit**. (For some signals under CMS, I/O also would not be available to handlers if they were called immediately.)

- Program reliability is improved by restricting the circumstances under which handlers can be called. For example, suppose your program adds elements to an array and contains the following code:

  ```
  ++elements;
  table[elements] = value;
  ```

  If a signal handler were called between these two statements, an attempt by a handler to add an element to the table would cause an entry to be skipped. Knowing that signals are not discovered at this point, you do not need to worry about skipped entries.

  Note: Combining the two statements into one does not improve the situation because signals can occur between any two machine instructions whether or not they are part of the same statement.

This method of discovering asynchronous signals is a feature of the SAS/C implementation. If you are writing portable code be aware that, on some systems, handlers are always called immediately. In such systems, you must write code carefully to avoid incorrect results if signals are inconveniently timed.

Delaying Discovery of Signals

For many applications, coding a signal handler is complicated by the possibility that a new signal may be generated during the handler's execution (either the same signal or a completely unrelated signal). For POSIX applications, the **sigaction** function makes it easy to block signals during the execution of a signal handler. For non-POSIX applications, the situation is more complicated, and it may be difficult to block asynchronous signals during the execution of a signal handler. To assist in the writing of reliable code, the SAS/C library suppresses the discovery of new asynchronous signals within a handler. An exception is when the handler calls **sigchk**, **signal**, or **sigaction**, which indicates its readiness to handle new signals. This applies only to asynchronous signals; for example, if a handler divides by 0, the resulting signal cannot be delayed no matter how convenient that might be. Also, any signals that are pending while a handler executes are discovered and processed when the handler returns.

If you are writing portable code be aware that, on some systems, asynchronous signals are discovered even while a handler is executing. On such systems, you must write code carefully to avoid incorrect results if signals are inconveniently timed.

Waiting for Signals

Some programs are *interrupt driven*; that is, their operation is controlled by signals from external sources (for example, IUCV signals from other VM users). For such programs, it is important to have a waiting period without using CPU resources until a signal is received. The library provides the following functions for this purpose:

- **pause**, **sigpause**, and **sigsuspend** suspend execution until a signal is received
- **ecbpause** and **ecbsuspend** wait for either a signal or for an Event Control Block (ECB) to be posted
- **sleep** and **sleepd** suspend execution until a signal is received or an elapsed time interval expires. If a signal is received while program execution is suspended by one of these functions, the signal is handled immediately, unless the signal is blocked.

Blocking Signals

Signals may be blocked to allow critical sections of code to run without interruption. The following functions control signal blocking:

`sigprockmask` and **`sigsuspend`**
: block either SAS/C managed signals or OpenEdition signals. **`sigprocmask`** and **`sigsuspend`** are portable to POSIX-conforming systems.

`sigblock`, **`sigsetmask`**, and **`sigpause`**
: are only used with SAS/C managed signals. These functions are portable to BSD UNIX operating systems.

In general, **`sigblock`**, **`sigsetmask`**, and **`sigpause`** are provided for compatibility with previous releases of SAS/C and code that will be ported to BSD UNIX operating systems. You are encouraged to use **`sigprockmask`** and **`sigsuspend`** when possible because they are more flexible, and they can handle signals managed by both SAS/C and OpenEdition.

Using sigprocmask

The **`sigprocmask`** function manipulates a signal mask that determines which signals are blocked. A blocked signal is not ignored; it is simply blocked from having any effect until the signal mask is reset. **`SIGKILL`** and **`SIGSTOP`** cannot be blocked by **`sigprocmask`**; otherwise, all signals managed by OpenEdition may be blocked. You can also use **`sigprocmask`** to block any of the asynchronous signals managed by SAS/C.

The **`sigprocmask`** function receives **`sigset_t`** structures for both a new and an old signal mask. **`sigprocmask`** also requires one of the following **`int`** arguments that specify how to use these signal masks.

`SIG_BLOCK`
: specifies that the signals in the new signal mask should be blocked.

`SIG_UNBLOCK`
: specifies that the signals in the new signal mask should not be blocked.

`SIG_SETMASK`
: specifies that the new mask should replace the old mask.

Here is an example that illustrates the **`sigprocmask`** function:

```
#include <sys/types.h>
#include <signal.h>
#include <lcsignal.h>
#include <stdlib.h>

main()
{
   sigset_t omvsManaged, sascManaged, pendingSignals, newMask, oldMask;


   sigfillset(&omvsManaged);        /* Request OpenEdition management */
   sigemptyset(&sascManaged);       /* of all signals.                */
   sigemptyset(&pendingSignals);
   sigemptyset(&newMask);
   sigemptyset(&oldMask);

      /* Tell the system which signals should be managed by SAS/C     */
      /*  and which by OpenEdition.                                    */
   oesigsetup(&omvsManaged, &sascManaged);
```

```
         /* Block the SIGHUP signal.                                   */
      sigaddset(&newMask, SIGHUP);
      sigprocmask(SIG_BLOCK, &newMask, &oldMask);

         /* Actions to take while SIGHUP is blocked go here.          */

         /* Check to see if SIGHUP is pending.                        */
      sigpending(&pendingSignals);
      if (sigismember(&pendingSignals, SIGHUP){

         /* Actions to take if SIGHUP is pending go here.             */

      }

         /* Restore the old mask.                                     */
      sigprocmask(SIG_SETMASK, &oldMask, NULL);
      .
      .
      .
   }
```

Notice that if you don't like the defaults value for signal management you must use **oesigsetup** to specify which signals are managed by OpenEdition before you can use **sigprocmask** to block a signal.

The **sigpending** function can be used to determine whether any signals are pending. A pending signal is a blocked signal that has occurred and for which the signal action is still pending. The **sigset_t** structure passed to **sigpending** is filled with the pending signals for the calling process. You can also use the **sigismember** function to determine if a specific signal is pending, as illustrated in the previous example.

If a call to **sigprocmask** causes the signal mask to be changed so that one or more blocked signals become unblocked, then at least one of these signals is delivered to the calling process before the return from **sigprocmask**. When the first signal is delivered, the signal mask changes according to the action specified for that signal; this might or might not reblock any remaining pending signals. If there are remaining pending signals blocked by the action for the first signal, then on return from the handler, the previous mask is restored and the next signal is delivered, and so on.

Note: The **sigsuspend** function also modifies the signal mask. See "Using sigsuspend and sigpause" on page 5-17 for more information.

Using sigblock, sigsetmask, and sigpause

The **sigblock**, **sigsetmask**, and **sigpause** functions can also be used to block signals managed by SAS/C. However, these functions cannot be used to block signals managed by OpenEdition.

Note: None of the facilities described in this section are completely portable. They are similar to facilities provided by the Berkeley 4.2 BSD UNIX implementation. The **sigprocmask** function described in the previous section provides a more flexible method of blocking signals.

Setting Up the Mask

The SAS/C library maintains a bit string called the *signal mask* that defines which asynchronous signals managed by SAS/C can be discovered and which of these signals are blocked. When a program begins execution no signals are blocked. If you set the mask to block a signal, the library does not discover an occurrence of the signal until that bit in the mask is reset. When you reset the mask to permit the signal to be discovered pending signals are then handled normally.

When you generate a signal with **raise** or **siggen**, the library permits the signal to be recognized even when it is blocked. Using the **sleep** function also permits the **SIGALRM** signal to be recognized even if it is blocked. In addition, the library may block signals temporarily to preserve the integrity of its data areas as it performs certain actions. When it has completed processing, the library restores the mask defined by your program.

The library allows any asynchronous signal managed by SAS/C (those defined by the library or **SIGASY1 through SIGASY8**) to be blocked by use of the functions **sigblock**, **sigsetmask**, and **sigpause**.

Note: Signal blocking is meaningful only for asynchronous signals. Calls to these functions for synchronous signals do not generate errors; the calls have no effect.

All of the signal-blocking functions require an argument that changes the bit string used to mask signals, but the effect of the argument differs among the functions. The **sigblock** function blocks the signals indicated by the argument but does not change the rest of the mask. **sigsetmask** and **sigpause** reset the mask so that only the signals indicated by the argument are blocked.

The form of the argument is the same for **sigblock**, **sigsetmask**, and **sigpause**. Use the left shift operator as shown here to specify a mask for a single signal:

```
1<<(signal - 1)
```

The following call to **sigblock** sets the mask to block interrupt signals in addition to any other masks already in effect:

```
/* Block SIGINT; retain rest of mask. */

sigblock(1<<(SIGINT - 1));
```

A similar call to **sigsetmask** changes the mask so that only interrupt signals are blocked:

```
sigsetmask(1<<(SIGINT - 1)); /* Block only SIGINT */
```

To block several signals with the **sigsetmask** function, use the bitwise OR operator. For example, this code blocks both the interrupt signal and the user-defined signal **SIGASY1**:

```
/* Block SIGINT and SIGASY1. */

sigsetmask(1<<(SIGINT - 1)|1<<(SIGASY1 - 1));
```

Using Signal Blocking

sigprocmask, **sigblock**, and **sigsetmask** are typically used to protect the execution of small sections of code that *must* run without interruption. To do this, use one of the signal-blocking functions to block interruptions before beginning the critical code. Then, when the critical actions are completed, restore the mask that was in effect before all signals were blocked. For example, the following code calls **sigprocmask** to block the additional signals identified by **blockedSignals** and store the previous mask in **oldMask**. Note that the new mask is the union of the current signal mask and the additional signals that are identified by **blockedSignals**. The example then calls **sigprocmask** a second time to restore the previous mask after the critical code is completed:

```
sigprocmask(SIG_BLOCK, &blockedSignals, &oldMask); /* Block signals. */
take_ckpt();                                       /* Checkpoint data.       */
sigprocmask(SIG_SETMASK, &oldmask, NULL);  /* Restore previous mask. */
```

In this example, you could instead use **sigsetmask** and **sigblock** if you were blocking only signals managed by SAS/C. Both **sigblock** and **sigsetmask** return the previous signal mask. For example, you could call **sigblock** as follows to block all signals and store the value of the old mask in **oldMask**:

```
oldMask = sigblock(0xffffffff);
```

Using sigsuspend and sigpause

Besides **sigprocmask**, **sigblock**, and **sigsetmask**, the library provides the **sigsuspend** and **sigpause** functions. These functions combine the actions of the **pause** and **sigprocmask** functions. Here is an example of a call to **sigsuspend**:

```
sigsuspend(&newMask);
```

The call to **sigsuspend** has approximately the same effect as the following code:

```
sigprocmask(SIG_SETMASK, &newMask, &oldMask);
pause();
sigprocmask(SIG_SETMASK, &oldMask, NULL);
```

sigpause is similar to **sigsuspend** except that its signal mask is in the BSD format rather than the POSIX **sigset_t** format, and it only allows you to change the blocking of signals managed by SAS/C.

There is one important difference between the call to **sigsuspend** and the sequence of the other function calls shown when **sigsuspend** is used: the second call to **sigprocmask** occurs before any handler is called. **sigsuspend** and **sigpause** are useful for interrupt-driven programs, which can perform necessary processing with all signals blocked and then pause with some or all signals unblocked when the program is ready to receive another signal. If a signal is pending, the handler for that signal is called; otherwise, the program waits until a new signal is received. In either case, the old mask, which blocked all signals, is restored before the handler is called. This ensures that no signals are discovered during execution of the handler, which guarantees that all signals are processed one at a time in the order they are received. Thus, processing of one signal is never interrupted by another. Programs that process only one signal at a time are more reliable and easier to write than those that permit interruption during most or all of their execution. Refer to the example in the description of **sigpause** on page 5-16 for an illustration of how to use these functions to process one signal at a time.

Signal blocking is useful even though, in many cases, the library suspends processing of other asynchronous signals while a signal handler is executing. Remember that the library allows processing of other asynchronous signals as soon as the signal handler calls **signal** to reinstate signal handling. Even if the handler issues the call to **signal** as the last instruction in the function, the handler may not complete execution before the next signal is discovered and handled.

Also, when you use **sigaction** to define a signal handler, you can use the arguments to **sigaction** to further control the blocking of signals within the handler. A signal is blocked during execution of a handler for a signal discovered during a call to **`sigsuspend`** or **`sigpause`** if it is blocked by either the signal mask in effect when **`sigsuspend`** or **`sigpause`** were called, or by the mask specified by **`sigaction`**.

Using Signals Portably

The SAS/C signal-handling implementation offers many extensions that make it easier to use than a completely standard implementation. However, if you use these extensions, your programs are not portable. Keep in mind these considerations when writing a program that you intend to be portable:

- Use only standard signals. These are **`SIGFPE`**, **`SIGSEGV`**, **`SIGILL`**, **`SIGINT`**, **`SIGABRT`**, and **`SIGTERM`**. Note that different implementations of C do not always generate signals for the same reasons. For example, in some implementations, a floating-point underflow may set the result to 0 instead of raising **`SIGFPE`**. The only signals that are guaranteed are those generated by calls to **`raise`** or **`abort`**.
- Of all the functions available with this implementation of C, only **`raise`** and **`signal`** are in the ANSI Standard library. The remaining functions, which may be convenient, cannot be used in a portable program. The functions **`sigaction`**, **`sigprocmask`**, **`sigpending`**, and **`sigsuspend`** are portable to other systems that implement the POSIX 1003.1 standard.
- A portable program must be able to deal with an interrupt at any time. A lot of code may be required to safely update linked lists and other data structures modified by the signal handler.
- A portable program that needs to handle a signal repeatedly should call **`signal`** again immediately after entering the handler. This call minimizes the chance that the same signal will occur again and cause program termination.

  Note: If you are writing programs to run only with the SAS/C library, you can delay calling **`signal`** until the end of the handler, because new asynchronous signals are not discovered until the handler returns, or **`signal`** is called. If you call **`signal`** at the beginning of the handler to reinstate signal handling, refer to the third list item.
- Not all implementations support using **`longjmp`** in a signal handler. Also, in implementations other than SAS/C, library data may be left in an inconsistent state if **`longjmp`** is called in a handler after a library function is interrupted. This can cause unpredictable results if the same function or a related function is called again. (The SAS/C library uses the **`blkjmp`** function in many cases to intercept **`longjmp`** from a handler, so the problem should not arise in a SAS/C program, except for signals associated with ABEND.)
- Most C library implementations are not re-entrant. For this reason, calling library functions other than **`longjmp`**, **`abort`**, **`exit`**, or **`signal`** from a handler is dangerous. This is especially true of memory allocation and I/O functions.
- Because signals can occur between machine instructions, assignments or tests may be only partially completed at the time a signal occurs. If a handler updates data that are being modified or tested at the time of the signal and returns, the effects are unpredictable.
- For maximum portability, your signal handlers should do the following:

  1. reinstate signal handling by calling **`signal`**.
  2. code the signal handler to assign a constant to a **`static volatile sig_atomic_t`** variable defined outside the handler. (The SAS/C library defines **`sig_atomic_t`** as **`char`**.)
  3. return.

Then, in an appropriate area of the main code, test the **`static`** variable set by the handler to determine whether a signal has occurred. Even using this process, incorrect results can occur if the compiler optimizes references to the variable. Making the variable **`volatile`** may help prevent such optimizations.

- All signal handlers should be defined to have the **`signum`** argument even if it is not used in the handler. Here is an example:

```
void int_handler(int signum)
```

Using Signals Reliably

The key to writing programs that handle signals reliably is to keep close control of when signals can occur. In particular, you can simplify program logic if you avoid handling signals while main-line code is testing or modifying variables that are also accessed or modified by a signal handler.

Some of the ways you can use the library facilities to control when signals must be handled are discussed here. Note that these techniques, with the exception of the first one, are specific to the SAS/C library.

- Code carefully to avoid generating computational signals, such as division by 0, or memory access violations. Because these signals cannot be delayed, the only way to keep them from occurring at an inconvenient time is to write code that does not cause them. Sometimes the input data make it impossible to avoid generating signals. In such situations, be aware that the signals can occur and be prepared to handle them.
- The library discovers asynchronous signals only when a function is called or returns. Therefore, manipulation of critical data is not interrupted by an asynchronous signal if you do not call any functions.
- If you must call a function to manipulate data tested or modified by a signal handler, call **`sigprocmask`** or **`sigblock`** before calling the function to minimize the chance of being interrupted. Then call **`sigprocmask`** or **`sigsetmask`** to restore the signal mask when the critical activity is completed. The call to **`sigprocmask`** or **`sigblock`** allows any pending signals to be discovered, so interruption is possible at this point. However, if no signals are pending, no other interruptions occur until you reset the mask with **`sigprocmask`** or **`sigsetmask`**.
- If you want to reinstate signal handling within a handler, delay calling **`signal`** in the handler for as long as possible. The library postpones processing new asynchronous signals while a handler is executing, unless the handler calls **`sigchk`** or **`signal`**. As soon as you call **`signal`**, any pending signals are discovered and handled. Alternately, use **`sigaction`** rather than **`signal`** to define your handlers. In this case, you do not need to call **`signal`** again to reinstate your handler, and you can have any signals you wish blocked automatically during the handler's execution.
- To ensure that interrupt-driven programs handle signals one at a time, use **`sigprocmask`** to block signals until the program is ready to process interruptions. When an interruption can be processed, call **`sigsuspend`** to wait for the next signal. When the signal occurs, the library automatically restores the pre-**`sigsuspend`** signal mask before calling your handler.

- □ Because the SAS/C library discovers signals only when you call a function or when it returns, you may need to add calls to **sigchk** to discover signals at points in your code that do not call any functions. Be sure to select points at which all data structures used by handlers are in a consistent state.
- □ If you have handlers that call the **longjmp** function, you may want to use the **blkjmp** function in routines that can be interrupted to protect some portions of the routine. This permits the routine to successfully complete activities that should not be interrupted. Note that if a handler calls **longjmp** while I/O is being performed, the error flag is set for the file. You must call **clearerr** to continue to use the file, but be aware that information may be lost from the file.
- □ Unless you block it, the **SIGINT** signal (as well as OpenEdition signals such as **SIGTTIN**) can occur at any time while the program is waiting for input from the terminal.

Signal Descriptions

This section provides a detailed description of each signal that can be handled by the SAS/C library, with the exception of **SIGIO**. (The **SIGIO** signal is not included because it currently has no special meaning for the SAS/C library.) Each description explains the information returned by a call to **siginfo** when a signal is generated naturally. When your program raises a signal by calling the **raise** function, a call to **siginfo** returns **NULL** except where noted otherwise. If the program raises a signal by calling **siggen**, **siginfo** returns the value of the second argument to **siggen**.

Note: The POSIX signals (**SIGCHLD**, **SIGCONT**, **SIGHUP**, **SIGKILL**, **SIGPIPE**, **SIGQUIT**, **SIGSTOP**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU**) are not included in this section because all pertinent information about them is contained in Tables 5.2 and 5.3 on pages 5-9 and 5-10, respectively. For general information about POSIX signal handling, refer to *The POSIX.1 Standard: A Programmer's Guide.*

SIGABND Program Abnormal Termination

The SIGABND signal is raised when the operating system terminates the C program with a system ABEND code. This ABEND either indicates a misuse of an operating system feature or an error by the system processing a valid request.

Default handling

By default, the signal causes abnormal termination of the program with the ABEND code requested by the operating system. If SIGABND results from a call to **siggen**, the ABEND code is taken from the **ABRT_t** structure passed to **siggen**.

Ignoring the signal

The **SIGABND** signal cannot be ignored. Similarly, a handler for **SIGABND** cannot return to the point of interrupt; an attempt to do so causes ABEND to be reissued.

Information returned by siginfo

If you call **siginfo** after a **SIGABND** signal occurs, **siginfo** returns a pointer to a structure of type **ABND_t**. This structure is defined as:

```
typedef struct {
   unsigned ABEND_code;    /* ABEND code            */
   char *ABEND_str;        /* formatted ABEND code,  */
                           /* e.g., "B14", "U0240",  */
   void *ABEND_info;      /* OS SDWA, or CMS ABWSECT */
} ABND_t;
```

The **ABEND_code** contains the unmodified system ABEND code. For example, after a system 106 ABEND, the ABEND_code value is 0x106. The **ABEND_str** is a null-terminated string giving a printable form of the ABEND code.

The **ABEND_info** pointer addresses an ABEND status block provided by the operating system, which gives the ABEND PSW, registers, and other such information. Under MVS, **ABEND_str** addresses an SDWA. Under CMS, it addresses an ABWSECT. For an ABEND issued by the SAS/C library, **ABEND_info** may be **NULL**.

Notes on defining a handler

A System ABEND in a C program is most likely to result from unexpected conditions in the C library. For example, the **fopen** function may ABEND if you attempt to open a file you are not authorized to access. If you provide a **SIGTERM** handler and use the **longjmp** function to continue program execution after such an ABEND, the library is not always able to clean up after the failure. This may prevent memory or other resources used by the library from being freed. For instance, after recovery from an ABEND in **fopen**, it may not be possible to open **FOPEN_MAX** files.

SIGABRT Program Abort

The **SIGABRT** signal is raised when the **abort** function is called or when a user ABEND occurs. **SIGABRT** may not be raised for an ABEND issued by the SAS/C library, depending on the severity of the problem.

Default handling

By default, **SIGABRT** causes abnormal program termination. If **SIGABRT** results from a call to **abort** or **raise**, the program is terminated with user ABEND code 1210. If **SIGABRT** results from a call to **siggen**, the ABEND code is taken from the **ABRT_t** structure passed to **siggen**.

Ignoring the signal

The **SIGABRT** signal cannot be ignored. Similarly, a handler for **SIGABRT** cannot return to the point of interrupt; an attempt to do so causes ABEND to be reissued.

Information returned by siginfo

If you call **siginfo** after a **SIGABRT** signal occurs, **siginfo** returns a pointer to a structure of type **ABRT_t**. This structure is defined as:

```
typedef struct {
   unsigned ABEND_code;   /* ABEND code              */
   char *ABEND_str;       /* formatted ABEND code,   */
                          /* e.g., "B14", "U0240",   */
   void *ABEND_info;      /* OS SDWA, or CMS ABWSECT */
} ABRT_t;
```

The **ABEND_code** is an integer from 0 through 4095 giving the ABEND code. The **ABEND_str** is a null-terminated string giving a printable form of the ABEND code.

The **ABEND_info** pointer addresses an ABEND status block provided by the operating system, which gives the ABEND PSW, registers, and other such information. Under MVS, **ABEND_str** addresses an SDWA. Under CMS, it addresses an ABWSECT. For an ABEND issued by the SAS/C library, **ABEND_info** may be **NULL**.

OpenEdition considerations

SAS/C uses **SIGABRT** to signal a user ABEND, including a library ABEND. This differs from OpenEdition, which expects every ABEND to be signaled by **SIGABND**.

SIGALRM Real-Time Expiration

`SIGALRM` is an asynchronous signal. The `SIGALRM` signal is raised when a time interval specified in a call to the `alarm` or `alarmd` function expires.

Because `SIGALRM` 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`. For this reason and because of inaccuracies and overhead in operating system timing functions, you can consider the time interval requested by `alarm` a lower bound; the handler may not be invoked immediately after the interval expires.

Default handling

By default, `SIGALRM` 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 `SIGALRM`.

Information returned by siginfo

When `siginfo` is called in a handler for `SIGALRM`, it returns `NULL`.

OpenEdition Considerations

If `SIGALRM` is managed by OpenEdition, the SAS/C `alarmd` and `sleepd` functions are not available. If the SAS/C library manages `SIGALRM`, the `ps` shell command will not accurately indicate when the process is sleeping.

SIGFPDIV Floating-Point Division by 0

The **`SIGFPDIV`** signal is raised when the second operand of the division operator (/) is 0, and default handling is in effect for **`SIGFPE`**. If you have specified a handler for **`SIGFPE`** (either **`SIG_IGN`** or a function you define), **`SIGFPDIV`** is not raised.

Default handling

If the **`SIGFPDIV`** signal is raised and default handling is in effect, the program abnormally terminates with an ABEND code of 0CF.

Ignoring the signal

If your program ignores **`SIGFPDIV`**, program execution continues, but the results of the failed expression are unpredictable.

Information returned by siginfo

If you call **`siginfo`** after a **`SIGFPDIV`** 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 **`SIGFPDIV`**, you can change the result of the computation by using the information returned by **`siginfo`**. Refer to the example in the description of the **`siginfo`** function for an illustration of this technique.

SIGFPE General Computational Error

The **SIGFPE** signal is raised when a computational error occurs. These errors include floating-point overflow, floating-point underflow, and either integer- or floating-point division by 0. Note that integer overflow never causes a signal; when integer overflow occurs, the result is reduced to 32 bits 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 for the **SIGFPOFL**, **SIGFPUFL**, **SIGFPDIV**, or **SIGIDIV** conditions. 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 underflows, the result is set to 0.

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 {
      int *intv;      /* result for integer expression           */
      double *doublev; /* result for double expression           */
      } result;
   char *EPIE;        /* pointer to hardware program check info */
   double *fpregs;    /* floating-point register contents        */
   } FPE_t;
```

The **int_code** field contains the number of the more specific signal associated with the **SIGFPOFL**, **SIGFPUFL**, **SIGFPDIV**, or **SIGIDIV** conditions. 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.

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 IBM publication *MVS/XA Supervisor Services and Macro Instructions.* (Although an EPIE is provided only by the XA versions of the MVS and CMS operating systems, one is created by the run-time library for all MVS and CMS systems.)

The **fpregs** field is a pointer to an array of doubles that contains the contents of the floating-point registers at the time of the signal and stored in the order 0, 2, 4, 6.

SIGFPE General Computational Error

(continued)

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.

OpenEdition Considerations

When `SIGFPE` is managed by OpenEdition, 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 `SIGFPDIV`, `SIGFPOFL`, `SIGFPUFL`, or `SIGIDIV` signals specific to SAS/C, you must define `SIGFPE` as a signal managed by SAS/C.

SIGFPOFL Floating-Point Overflow

The **SIGFPOFL** signal is raised when the magnitude of the result of a floating-point computation exceeds the maximum supported by the hardware and default handling is in effect for **SIGFPE**. If you have specified a handler for **SIGFPE** (either **SIG_IGN** or a function you define), **SIGFPOFL** is not raised.

Default handling

If the **SIGFPOFL** signal is raised and default handling is in effect, the program abnormally terminates with an ABEND code of 0CC.

Ignoring the signal

If your program ignores **SIGFPOFL**, program execution continues, but the results of the failed expression are unpredictable.

Information returned by siginfo

If you call **siginfo** after a **SIGFPOFL** 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 **SIGFPOFL**, you can change the result of the computation by using the information returned by **siginfo**. Refer to the example in the description of the **siginfo** function for an illustration of this technique.

SIGFPUFL Floating-Point Underflow

The **`SIGFPUFL`** signal is raised when the magnitude of the nonzero result of a floating-point computation is smaller than the smallest nonzero floating-point value supported by the hardware and default handling is in effect for **`SIGFPE`**. If you specified a handler for **`SIGFPE`** (either **`SIG_IGN`** or a function you define), **`SIGFPUFL`** is not raised.

Default handling

If the **`SIGFPUFL`** signal is raised and default handling is in effect, the result of the computation that raised the signal is set to 0 and execution continues normally.

Ignoring the signal

If your program ignores **`SIGFPUFL`**, program execution continues, and the result of the computation that raised the signal is set to 0.

Information returned by siginfo

If you call **`siginfo`** after a **`SIGFPUFL`** 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 **`SIGFPUFL`**, you can change the result of the computation by using the information returned by **`siginfo`**. Refer to the example in the description of the **`siginfo`** function for an illustration of this technique.

Note: If a handler defined for **`SIGFPUFL`** does not change the value of the result, the result of the computation is undefined and it is not automatically set to 0.

SIGIDIV Integer Division by 0

The `SIGIDIV` signal is raised when the second operand of a division operator (/ or %) is 0 and default handling is in effect for `SIGFPE`. If you have specified a handler for `SIGFPE` (either `SIG_IGN` or a function you define), `SIGIDIV` is not raised.

Default handling

If the `SIGIDIV` signal is raised and default handling is in effect, the program abnormally terminates with an ABEND code of 0C9.

Ignoring the signal

If your program ignores `SIGIDIV`, program execution continues but the result of the computation that raised the signal is undefined.

Information returned by siginfo

If you call `siginfo` after a `SIGIDIV` 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 `SIGIDIV`, you can change the result of the computation by using the information returned by `siginfo`. Refer to the example in the description of the `siginfo` function for an illustration of this technique.

SIGILL Illegal Instruction

The **SIGILL** signal is raised when an attempt is made to execute an invalid, privileged, or ill-formed instruction. **SIGILL** is usually caused by a program error that overlays code with data or by a call to a function that is not linked into the program load module.

Default handling

By default, **SIGILL** causes program termination with an appropriate ABEND code (0C1 for an operation error, 0C2 for a privileged operation error, 0C3 for an execute error, or 0C6 for a specification error).

Ignoring the signal

The **SIGILL** signal cannot be ignored. If you code **SIG_IGN** as the second argument to **signal**, the call to **signal** is rejected.

Information returned by siginfo

If you call **siginfo** in a handler for **SIGILL**, **siginfo** returns a pointer to a structure of type **ILL_t**. This structure is defined as:

```
typedef struct {
   int int_code;  /* interrupt code                          */
   char *EPIE;    /* pointer to hardware program check info */
   } ILL_t;
```

The **int_code** field of this structure contains the program code indicating what type of illegal instruction occurred. Refer to "Default handling" above.

The **EPIE** field is a pointer to a control block containing hardware information available at the time the signal occurred. (This information includes program status word and registers.) For information on the EPIE format, see IBM publication *MVS/XA Supervisor Services and Macro Instructions.* (Although an EPIE is provided only by the XA versions of the MVS and CMS operating systems, one is created by the run-time library for all MVS and CMS systems.)

Notes on defining a handler

If you define a handler for **SIGILL**, you can call **siginfo** and test the **int_code** field of the structure (returned by a call to **siginfo**) to determine what error occurred. Note that a handler for **SIGILL** cannot return to the point of interrupt; an attempt to do so causes the program to terminate as described in "Default handling" above.

SIGINT Interactive Terminal Attention Signal

SIGINT is an asynchronous signal. The **SIGINT** signal is raised when the terminal user requests a program interruption. Under MVS, the terminal PA1 or ATTN key raises the **SIGINT** signal; under CMS, the IC (Interrupt C) immediate command raises **SIGINT**. However, if you are executing the program using the debugger, you must use the debugger **attn** command to generate a **SIGINT** signal. (The PA1/ATTN key or the IC command is intercepted and handled by the debugger.) The debugger **attn** command is handled as if **SIGINT** were raised by the normal methods.

Default handling

The library does not perform any default actions for the **SIGINT** signal. If the program is executing under MVS, the PA1/ATTN key is handled by the program that invoked the C program (for example, ISPF or the TSO terminal monitor program). If the program is executing under CMS, the IC command is treated as an unknown command.

Ignoring the signal

Ignoring **SIGINT** by coding **SIG_IGN** as the second argument in the call to **signal** does not have the same effect as default handling. If **SIGINT** is ignored, use of the PA1/ATTN key or the IC command is recognized but it has no effect on the program.

Information returned by siginfo

When **siginfo** is called in a handler for **SIGINT**, it returns **NULL**.

Notes on defining a handler

Because **SIGINT** is an asynchronous signal, the library discovers the signal only when you call a function, when a function returns, or when you issue a call to **sigchk**. **SIGINT** frequently occurs while the program is reading from the terminal. If this occurs and the handler for **SIGINT** returns to the point at which the signal occurred, the input request is reissued, unless the handler set the error flag for the file.

SIGIUCV CMS Inter-User Communication

The `SIGIUCV` signal is raised as a result of communication between two VM users. The `SIGIUCV` signal can be generated only for programs that have used the `iucvset` function to initialize communication.

`SIGIUCV` is an asynchronous signal. For this reason, a handler for `SIGIUCV` can only be invoked when a function is called or returns, or when `sigchk` is used.

Default handling

By default, `SIGIUCV` causes the program to abnormally terminate with a user ABEND code of 1225. For this reason, you must have a `SIGIUCV` handler defined at all times that a signal can be discovered, if your program uses `SIGIUCV`.

Ignoring the signal

The `SIGIUCV` signal cannot be ignored. If you code `SIG_IGN` as the second argument to `signal` and an IUCV signal is received, the program terminates, as described in “Default handling” above.

Information returned by siginfo

If you call `siginfo` in a handler for `SIGIUCV`, it returns a pointer to a structure of one of several types, depending on the particular interrupt. This structure contains information about the communication that caused the signal. For example, if the signal indicates that a message has been sent by another user, you can call the `iucvrecv` function to obtain the message text. Refer to Chapter 5, “Inter-User Communications Vehicle (IUCV) Functions,” in *SAS/C Library Reference, Volume 2* for more information on what is returned by `siginfo`.

Notes on raising SIGIUCV

Use of `raise` or `siggen` with `SIGIUCV` has no effect on the status of any pending signals. Signals generated with `raise` are always synchronous; that is, they are never delayed, even if blocked, so an artificially generated `SIGIUCV` signal may be handled before any pending real `SIGIUCV` signals.

SIGMEM No Memory Available for Stack Space

The `SIGMEM` signal is raised when a function call requires additional stack space, but little space is available. At the start of program execution, 4K of stack space is reserved for emergency use; when no other space is available, the `SIGMEM` signal is raised. The reserved stack space is available to ensure that you can still execute a handler for the `SIGMEM` signal.

Note: If you use the `=minimal` run-time option to suppress stack overflow checking, the `SIGMEM` signal does not occur. In this case, if you run out of stack space, your program will probably ABEND as it tries to write past the end of the stack.

Default handling

The default handling for the `SIGMEM` signal is to ignore the condition. If the program can finish executing in the 4K of stack space reserved for emergency use, the program completes normally. If the program requires more than the emergency allocation, the program abnormally terminates with an 80A ABEND in MVS or a 0F7 in CMS.

Ignoring the signal

If your program ignores `SIGMEM`, processing proceeds as described in "Default handling" above.

Information returned by siginfo

If you call `siginfo` after a `SIGMEM` signal occurs, `siginfo` returns a pointer to an integer that contains the number of bytes required. This is only an approximation, and there is no guarantee that freeing this amount of memory will permit the failed allocation to succeed.

Notes on defining a handler

Because a `SIGMEM` handler is called when there is little memory available, you should avoid using any functions that require large amounts of memory. In particular, avoid opening files in a `SIGMEM` handler. You also should avoid output to `stdout` or `stderr`, unless these files have been used because these files are only partially open until they are first used. If you want to handle `SIGMEM` by writing an error message and terminating and you cannot guarantee that you have already used the diagnostic file, use `longjmp` to exit from the handler and write the message on completion of the jump. Termination of intermediate routines by `longjmp` may cause additional stack space to become available.

If the handler for `SIGMEM` returns to the point of interrupt, another attempt is made to allocate more stack space. If this attempt fails, the emergency allocation is used if less than 4K is required. If the emergency space is not sufficient, the program abnormally terminates.

After `SIGMEM` is raised during a program's execution, it is not raised again until one or more stack allocations have been successfully performed. This avoids the possibility of endless loops in which `SIGMEM` is raised repeatedly.

SIGSEGV Memory Access Violation

The **SIGSEGV** signal is raised when you attempt to illegally access or modify memory. **SIGSEGV** is usually caused by using uninitialized or **NULL** pointer values or by memory overlays.

Default handling

By default, **SIGSEGV** causes program termination with an appropriate ABEND code (0C4 for a protection error or 0C5 for an addressing error).

Ignoring the signal

The **SIGSEGV** signal cannot be ignored. If you code **SIG_IGN** as the second argument to **signal**, the call to **signal** is rejected.

Information returned by siginfo

If you call **siginfo** in a handler for **SIGSEGV**, **siginfo** returns a pointer to a structure of type **SEGV_t**. This structure is defined as:

```
typedef struct {
   int int_code;  /* interrupt code                        */
   char *EPIE;    /* pointer to hardware program check info */
   } SEGV_t;
```

The fields in this structure are the same as those in the structure type **ILL_t**; refer to the description of **SIGILL** for details on this structure.

Notes on defining a handler

If you define a handler for **SIGSEGV**, you can call **siginfo** and test the **int_code** field of the structure (returned by a call to **siginfo**) to determine what error occurred. A handler for **SIGSEGV** cannot return to the point of interrupt; an attempt to do so causes the program to terminate, as described in "Default handling" above.

Note: If the program overlays library control blocks, the **SIGSEGV** signal may cause an immediate unrecoverable program ABEND, even when a signal handler has been defined.

SIGTERM Termination Request

The `SIGTERM` signal can only be generated by a call to either `raise` or `siggen` when `SIGTERM` is managed by SAS/C.

Default handling

By default, the signal causes abnormal termination of the program with a user ABEND code of 1225.

Ignoring the signal

If your program ignores `SIGTERM`, program execution proceeds.

Information returned by siginfo

When `siginfo` is called in a handler for `SIGTERM`, it returns `NULL`.

Part 2

Function Reference

6 Function Descriptions

Introduction

Descriptions of the functions listed in Chapter 2, “Function Categories” on page 2-1 are provided in this chapter. Each description includes a synopsis, description, discussions of return values and portability issues, and an example. Also, errors, cautions, diagnostics, implementation details, and usage notes are included, if appropriate. Unless stated otherwise, any Standard function is also defined by traditional UNIX C compilers. See Chapter 1, “Introduction to the SAS/C Library” on page 1-1 for a more complete description of this book’s definition of UNIX.

abend Abnormally Terminate Execution Using ABEND

SAS/C

SYNOPSIS

```
#include <lclib.h>

void abend(int code);
```

DESCRIPTION

abend terminates program execution using the ABEND macro. The **code** argument is used as a user ABEND code. If **code** is not between 0 and 4095, the actual ABEND code is unpredictable. Open files are not closed, and **atexit** routines are not called before termination.

The SIGABRT signal is raised before termination occurs. Program execution continues if the signal handler exits using **longjmp**. If SIGABRT is managed by OpenEdition software rather than by SAS/C software, the program's final termination status is "terminated by SIGABRT." Also, the ABEND code is not directly accessible to a program running under the shell.

RETURN VALUE

Control is never returned to the caller of **abend**.

CAUTION

If you call **abend** without closing files, data in the files may be lost. In addition, an open UNIX style output file that requires copying is unchanged if you call **abend**. (See Chapter 3, "I/O Functions" on page 3-1 for more information on UNIX style files.)

EXAMPLE

```
#include <lclib.h>
#include <stdio.h>

void fatal(int errcode)
{
      /* Terminate execution after fatal error. */
   fprintf(stderr, "Fatal error %d, terminating.\n",
           errcode);
   fclose(stderr); fclose(stdout);
      /* Close standard files before ABEND.     */
   abend(errcode);
}
```

RELATED FUNCTIONS

abort

SEE ALSO

"SIGABRT" on page 5-22

"Program Control Functions" on page 2-9

abort Abnormally Terminate Execution

POSIX ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

void abort(void);
```

DESCRIPTION

abort terminates program execution abnormally. Open files are not closed before termination.

RETURN VALUE

Control is never returned to the caller of **abort**.

CAUTION

If you call **abort** without closing files, data in the files may be lost. In addition, an open UNIX style output file will be unchanged if you call **abort**. See Chapter 3, "I/O Functions" on page 3-1 for a definition of a *UNIX style file.*

IMPLEMENTATION

abort terminates by raising the SIGABRT signal. If a handler is not defined for this signal, SIGABRT causes program termination with a user ABEND code of 1210. See Chapter 5, "Signal-Handling Functions" on page 5-1 for more information about the SIGABRT signal.

EXAMPLE

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

main()
{
   char *passwd, reply[40];
   passwd = "SAS";

   puts("Passwd? ");
   gets(reply);

   if (strcmp(passwd,reply)){
     puts("Password incorrect; execution terminating abnormally.");
     fclose(stdout);
     abort();
   }

   puts("Password is correct.");
}
```

RELATED FUNCTIONS

abend

abort *(continued)*

Abnormally Terminate Execution

SEE ALSO

"SIGABRT" on page 5-22

"Program Control Functions" on page 2-9

abs Integer Conversion: Absolute Value

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>
int abs(int y);
```

DESCRIPTION

abs returns the absolute value of an integer.

RETURN VALUE

abs returns the absolute value of its argument.

IMPLEMENTATION

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

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>

#define BASELINE 32

main()
{
   int  range, temp;
   puts("The average temperature in NY in December is 32 degrees.");
   puts("Enter the average temperature in NC in December:");
   scanf("%d", &temp);

   range = abs(BASELINE - temp);    /* Calculate range. */

   printf("The average temperatures differ by: %d\n", range );
}
```

RELATED FUNCTIONS

fabs, **labs**

SEE ALSO

"Mathematical Functions" on page 2-6

access Test for File Existence and Access Privileges

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int access(const char *path, int amode);
```

SAS/C enables the header file **`<fcntl.h>`** to be included in place of **`<unistd.h>`**.

DESCRIPTION

The **`access`** function determines if a file exists and if that file can be read or written. **`path`** points to a filename of any style. The type of access tested for is given by **`amode`**, which can be the sum of one or more of the following symbols:

`F_OK`	indicates that the file exists.
`R_OK`	indicates read access.
`W_OK`	indicates write access.
`X_OK`	indicates execute access.

Use the bitwise inclusive OR to test several access modes simultaneously. You cannot use OR to specify **`F_OK`** with another symbol. **`X_OK`** is ignored, except for OpenEdition hierarchical file system (HFS) files.

RETURN VALUE

The **`access`** function returns 0 if the file exists and if the type (or types) of access specified by **`amode`** is allowed. If the file does not exist or the specified type of access is not allowed, −1 is returned.

CAUTION

MVS

A sequential file exists if it contains any data (as recorded in the VTOC). For more information on MVS file existence, see Chapter 3, "I/O Functions" on page 3-1.

A 0 return value from **`access`** does not necessarily mean that the file can be opened, even if the type of access is allowed. For example, a file may fail to open because incorrect DCB information is specified. A −1 return value always indicates that the file cannot be opened with the specified type of access.

CMS

If the filemode is not specified, **`*`** is used, unless **`amode`** indicates that write access is to be tested. In this case, **`A1`** is used as the filemode.

If the filename is in **`xed`** style and XEDIT is not active or the file is not found in XEDIT, the file is searched for on disk. Write access (**`W_OK`**) is not allowed for **`xed`** or **`sfd`** style files.

Under CMS, the **`access`** function cannot be used with VSAM files.

access Test for File Existence and Access Privileges
(continued)

IMPLEMENTATION

Any **amode** value can be tested for any device type. If the mode is not valid for a device, −1 is returned.

EXAMPLES

```
#include <fcntl.h>
#include <stdio.h>

main()
{
   int rc;

      /* Does the program have read and write access to the */
      /*  TSO ISPF profile dataset?                         */
   rc = access("tso:ispf.profile", R_OK + W_OK);

   if (rc == 0)
      puts("Read and write access exists.");
   else
      puts("Read and write access does not exist.");

      /* Does the member DATA1 exist in the partitioned     */
      /*  dataset referred to by the ddname MYPROG?         */
   rc = access("tso:myprog(data1)", 0);

   if (rc == 0)
      puts("File exists.");
   else
      puts("File does not exist.");

      /* Can SYS1.PARMLIB be updated?                       */
   rc = access("dsn:sys1.parmlib", W_OK);

   if (rc == 0)
      puts("Yes, SYS1.PARMLIB can be updated.");
   else
      puts("SYS1.PARMLIB cannot be updated.");
}
```

RELATED FUNCTIONS

cmsstat, **stat**

SEE ALSO

"File Management Functions" on page 2-16

_access Test for HFS File Existence and Access Privileges

DESCRIPTION

`_access` is used exactly like the standard `access` function. The argument to `_access` is interpreted as an HFS filename, even if it appears to begin with a style prefix or a leading // or both. `_access` runs faster and calls fewer other library routines than `access`. See `access` for a full description.

acos Compute the Trigonometric Arc Cosine

SYNOPSIS

```
#include <math.h>

double acos(double x);
```

DESCRIPTION

acos computes the trigonometric arc cosine of the argument **x**. The arc cosine function is the inverse of the trigonometric cosine function and is expressed by the following relation:

$$r = \cos^{-1}(x)$$

x is in the closed interval [−1.0,1.0].

RETURN VALUE

acos returns the principal value of the arc cosine of the argument **x**, provided that this value is defined and computable. The return value is a double precision floating-point number in the closed interval $[0,\pi]$ radians.

DIAGNOSTICS

An error message is written to the standard error file (**stderr**) by the run-time library if **x** is not in the domain [−1.0,1.0]. In this case, the function returns 0.0.

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

EXAMPLE

This example computes the value of pi:

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

main()
{
   double pi;

   /* pi is equal to the arc cosine of .5 times 3. */
   pi = 3 * acos(.5);

   printf("3 * acos(%f) = %f\n", .5, pi);
}
```

acos *(continued)* Compute the Trigonometric Arc Cosine

RELATED FUNCTIONS

`asin`, **`atan`**, **`_matherr`**

SEE ALSO

"Mathematical Functions" on page 2-6

afflush Flush File Buffers to Disk

SYNOPSIS

```
#include <lcio.h>

int afflush(FILE *f, int toeof);
```

DESCRIPTION

The **afflush** function flushes the output buffers for the **FILE** addressed by **f** to disk and performs additional system-dependent operations to ensure that the data will be accessible later, even if the program or the system fails. If the **toeof** argument is nonzero, the file is positioned at the end of the file as buffers are flushed; otherwise, the file position remains unchanged. You can use the **afflush** function with **FILE** opened for keyed access as well for text or binary access.

For files opened with the **trunc=yes** amparm specified or defaulted, **afflush** truncates the file at the current file position; that is, all characters after the current position are erased. If this behavior is not desired, a nonzero **toeof** argument can be specified to position to the end of file, thereby avoiding truncation.

For nondisk files, such as the terminal, **afflush** is treated the same as **fflush**, preceded by positioning at the end of the file if the **toeof** argument is nonzero.

The **afflush** function fails if the last file operation was a read. On completion of **afflush**, the next operation can be either a read or a write.

RETURN VALUES

The **afflush** function returns 0 or **EOF** if an error occurs.

CAUTION

For some file types, **afflush** may be rejected if the file is not positioned to the end of a record. This restriction applies to PDS members opened with **grow=no**.

Using **afflush** is significantly more costly than **fflush**.

IMPLEMENTATION

After flushing buffers, **afflush** ensures file integrity under MVS by issuing the CLOSE TYPE=T macro. Under CMS, it issues FSCLOSE for standard CMS disk files, followed by a call to FINIS for the associated minidisk, or it calls DMSCOMM (shared file commit) for CMS shared files. For OpenEdition HFS files, it invokes the **fsync** system call.

afflush Flush File Buffers to Disk
(continued)

EXAMPLES

```
#include <lcio.h>
#include <stdlib.h>

extern int num_updates;
extern FILE *database;

main()
{
   int rc;
   extern int num_updates;
   extern FILE *database;
   int transaction(FILE *); /* Update database, return number of  */
                            /*  transactions or negative to quit. */

   database = fopen("dsn:userid.DATABASE", "r");

   num_updates = 0;         /* Reset update counter.              */
   for(;;) {     /* Run transactions until quitting time.         */
      int trans;
      trans = transaction(database);
         /* Perform the next transaction.                         */
      if (trans < 0) break;
      num_updates += trans; /* Monitor number of updates.         */
      if (num_updates >= 100) { /* Every 100 updates, checkpoint. */
          rc = afflush(database, 0);  /* Flush updates to disk.   */
          if (rc != 0) {
             puts("Error saving recent updates.");
             fclose(database);
             abort();
          }
      }
   }

   fclose(database);
}
```

RELATED FUNCTIONS

fflush, **fsync**

SEE ALSO

“I/O Functions” on page 2-13

afopen Open a File with System-Dependent Options

SYNOPSIS

```
#include <lcio.h>

FILE *afopen(const char *name, const char *mode,
             const char *am, const char *amparms);
```

DESCRIPTION

The **afopen** function is an augmented version of the standard I/O **fopen** function. It enables the specification of various implementation-dependent and system-dependent parameters.

The **name** argument is the external name (sometimes called a *pathname*) of the file to be opened. Its form is operating-system- dependent. See “General filename specification” on page 3-28 for syntax details.

The **mode** argument is a string defining how the file will be used. The **mode** string contains one to three characters with the following syntax:

```
r  | [b]
w  | [+] | [k]
a
```

r	indicates that the file will be read.
w	indicates that the file will be written and the previous contents discarded.
a	indicates that data will be added to the end of the file. (If **a** is specified, the file is automatically positioned at the end of the file immediately before data are physically written.) For a keyed file, new data can be added anywhere in the file, but existing records cannot be changed.
+	indicates that the file will be both read and written if it is present in the mode string.
b	requests access as a binary stream. If you specified neither **b** nor **k**, text access is assumed. **k** requests access as a keyed stream.

The **am** argument is a string naming an access method. Remember to enclose the method in quotes. The following specifications are permitted:

``''	enables the library to select the access method.
term	applies only to terminal files.
The **seq** access method	is primarily oriented toward sequential access. It can be specified for terminal files, in which case, the **term** access method is automatically substituted.
rel	is primarily oriented toward access by relative character number. You can use the **rel** access method only when the open mode specifies binary access.

afopen Open a File with System-Dependent Options
(continued)

kvs	is oriented toward access to data by physical or logical keys. The **kvs** access method can be used only with VSAM files and only when the mode specifies keyed access.
fd	provides access to OpenEdition HFS files.

If the file cannot be handled with the access method you specify, the **afopen** operation fails.

See "Library access method selection" on page 3-35 for more information on access method selection.

The **amparms** argument is a string specifying access method parameters, which are system- and access-method-dependent file processing options. See "Access method parameters" on page 3-36 for a complete description of amparms.

RETURN VALUE

If successful, **afopen** returns a **FILE** object associated with the named file. If the file cannot be opened, a **NULL** value is returned.

IMPLEMENTATION

You can use files opened with **afopen** and files opened with **fopen** interchangeably. The **name** and **mode** arguments to **afopen** have the same meanings and formats as the corresponding **fopen** arguments.

The function call **afopen(name, mode, \`\`'', \`\`'')** is equivalent to **fopen(name, mode)**.

EXAMPLES

This example saves an output matrix in a file of type **matrix** with 10 rows of the matrix in each block of the file. If the file does not exist, create it with enough space for the matrix.

```
#include <lcio.h>
#include <stdio.h>

main()
{
   FILE *matrix_out;
   char *matrix_name;
   char path_name[50];

      /* matrix parameters for afopen                */
   char matrix_parms[90];
   int  matrix_rows;  /* number of rows in the matrix */
   double *matrix;    /* matrix definition           */

   matrix_name = "square";
   sprintf(path_name,"tso:%s.matrix", matrix_name);
```

afopen Open a File with System-Dependent Options
(continued)

```
      /* Set the afopen parameters.                   */
   sprintf(matrix_parms,
        "recfm=f, reclen=%d, blksize=%d, alcunit=block,"
        "space=%d", matrix_rows*sizeof(double),
        matrix_rows*sizeof(double)*10,
        matrix_rows / 10 + 1);

   matrix_out = afopen(path_name, "wb", "", matrix_parms);

     /* Write the entire matrix out at once.          */
   if (matrix_out)
      fwrite((char *)matrix, sizeof(double),
             matrix_rows*matrix_rows, matrix_out);
   else
      puts("Matrix file failed to open.");
}
```

RELATED FUNCTIONS

`aopen`, **`fopen`**

SEE ALSO

"Opening Files" on page 3-27

"Access method parameters" on page 3-36

"I/O Functions" on page 2-13

afread Read a Record

SYNOPSIS

```
#include <lcio.h>

size_t afread(void *ptr, size_t size, size_t count, FILE *f);
```

DESCRIPTION

afread reads items from the stream associated with the **FILE** object addressed by **f** until a record break is encountered. **size** defines the size of each item, **count** defines the maximum number of items to be read, and **ptr** addresses the area into which the items will be read. If the current record contains more than **count** items, a diagnostic message is generated and the file's error flag is set.

Calls to **afread** to obtain items of type **typeval** commonly have this form:

```
typeval buf[count];
afread(buf, sizeof(typeval), count, f);
```

afread is supported only for binary streams. You can use the **fgets** function to read a record from a text stream. See "Augmented Standard I/O" on page 3-63 for more information on **afread**.

RETURN VALUE

afread returns the number of items read from the record (which may be less than the maximum).

CAUTION

When used on a file with relative attributes, **afread** behaves exactly like **fread** because these files are processed as a continuous stream of characters without record boundaries. To process a file with relative attributes on a record-by-record basis, you must open it with **afopen** and specify the **''seq''** access method.

If **afread** reads a zero-length record, it skips it and ignores it. Use the **afread0** function if you are processing a file that may contain zero-length records.

DIAGNOSTICS

afread never reads past the end of the current record; an error occurs if the record contains a fractional number of items or if it contains more data after **count** items.

The return value from **afread** does not indicate whether the call was completely successful. You can use the **ferror** function to determine whether an error occurred.

afread Read a Record
(continued)

EXAMPLE

This example copies a single record from one file to another.

```
#include <stdio.h>

main()
{
   FILE *input, *output;
   char buf[500];
   int len;

      /* Open file with undefined length records.   */
   input = afopen("tso:INPUT", "rb", "seq",
                  "recfm=u, reclen=50");
   output = afopen("tso:WRITE", "wb","seq", "");

      /* Read a record--len contains record length. */
   len = afread(buf, 1, 50, input);

   afwrite(buf, 1, len, output);

   fclose(input);
   fclose(output);
}
```

RELATED FUNCTIONS

afread0, **afreadh**, **fgets**, **kretrv**

SEE ALSO

"Augmented Standard I/O" on page 3-63

"I/O Functions" on page 2-13

afread0 Read a Record (Possibly Length 0)

SYNOPSIS

```
#include <lcio.h>

int afread0(void *ptr, size_t size, size_t count, FILE *f);
```

DESCRIPTION

afread0 reads items from the stream associated with the **FILE** object addressed by **f** until a record break is encountered. **size** defines the size of each item, **count** defines the maximum number of items to be read, and **ptr** addresses the area into which the items will be read. If the current record contains more than **count** items, a diagnostic message is generated and the file's error flag is set. A zero-length record is considered to be a valid record containing 0 items.

Calls to **afread0** to obtain items of type **typeval** commonly have the form

```
typeval buf[count];
numread = afread0(buf, sizeof(typeval), count, f);
```

afread0 is supported only for binary streams. You can use the **fgets** function to read a record from a text stream. See "Augmented Standard I/O" on page 3-63 for more information on **afread0**.

Note: **afread0** differs from **afread** only in the type of the return value and the treatment of zero-length records.

RETURN VALUE

afread0 returns the number of items read from the record (which may be less than the maximum or zero). If an error or end-of-file occurs, a negative value is returned.

CAUTION

When used on a file with relative attributes, **afread0** behaves exactly like **fread** because these files are processed as continuous streams of characters without record boundaries. To process a file with relative attributes on a record-by-record basis, you must open it with **afopen** and specify the **''seq''** access method.

DIAGNOSTICS

afread0 never reads past the end of the current record; an error occurs if the record contains a fractional number of items or if it contains more data after **count** items.

The return value from **afread0** does not distinguish between end of file and an error condition. Use the **ferror** function to make this distinction.

EXAMPLE

This example copies one file to another, preserving the record structure and including zero-length records. (It may not work on LRECL=X files because the record size for this kind of file is unbounded.) The input and output arguments are given as command line arguments.

```
#include <lcio.h>
```

afread0 Read a Record (Possibly Length 0)
(continued)

```
#include <stdlib.h>

char *_style = "tso";   /* Assume tso-style file names.         */

main(int argc, char *argv[]) {
   FILE *in, *out;
   char *buf;           /* will be allocated to hold one record */
   int recsize;
   int count;
   int rc;
   if (argc < 3) {
      puts("Two arguments are required.");
      exit(EXIT_FAILURE);
   }
   if (argc > 3)
      puts("Extraneous command line arguments ignored.");
   in = fopen(argv[1], "rb");
   out = fopen(argv[2], "wb");
   if (!in || !out) {
      puts("File(s) failed to open.");
      exit(EXIT_FAILURE);
   }
      /* Get input file record size.   */
   recsize = fattr(in)->reclen;
      /* Guess if record size unknown. */
   if (recsize == 0) recsize = 65536;

      /* Allocate a buffer area.       */
   buf = malloc(recsize);
   if (buf == NULL) exit(EXIT_FAILURE);
   for(;;) {
         /* Read a record.     */
      count = afread0(buf, 1, recsize, in);
         /* EOF or input error */
      if (count < 0 || ferror(in)) break;
         /* Write the record.  */
      count = afwrite0(buf, 1, count, out);
         /* output error       */
      if (count < 0 || ferror(out)) break;
   }
   if (ferror(in) || ferror(out)) rc = EXIT_FAILURE;
   else rc = EXIT_SUCCESS;
   if (rc == EXIT_SUCCESS)
      puts("Copy was successful.");
   else puts("Copy failed (see library messages).");
   fclose(in);
   fclose(out);
   exit(rc);
}
```

afread0 Read a Record (Possibly Length 0)
(continued)

RELATED FUNCTIONS

`afread`

SEE ALSO

"Augmented Standard I/O" on page 3-63

"I/O Functions" on page 2-13

afreadh Read Part of a Record

SYNOPSIS

```
#include <lcio.h>

size_t afreadh(void *ptr, size_t size, size_t count, FILE *f);
```

DESCRIPTION

afreadh reads up to **count** items from the current record of the stream associated with the **FILE** object that **f** addresses. **size** defines the size of each item, and **ptr** addresses the area into which the items will be read.

Calls to **afreadh** to obtain items of type **typeval** commonly have this form:

```
typeval buf[count];
afreadh(buf, sizeof(typeval), count, f);
```

afreadh can only be used with a binary stream. See "Augmented Standard I/O" on page 3-63 for more information.

RETURN VALUE

afreadh returns the number of items read from the record (which may be less than the maximum).

CAUTION

When used on a file with relative attributes, **afreadh** behaves exactly like **fread** because these files are processed as a continuous stream of characters without record boundaries. To process a file with relative attributes on a record-by-record basis, you must open it with **afopen** and specify the **''seq''** access method.

If **afreadh** reads a zero-length record, it skips it and ignores it. Use the **afread0** function if you are processing a file that may contain zero-length records.

DIAGNOSTICS

afreadh never reads past the end of the current record; an error occurs if the record contains a fractional number of items.

The return value from **afreadh** does not indicate whether the call was completely successful. You can use the **ferror** function to determine whether an error occurred.

EXAMPLE

```
#include <lcio.h>
#define NAMESIZE 30
#define ADDRSIZE 80

main()
{
   FILE *custf;
   struct hdr {
```

afreadh Read Part of a Record
(continued)

```
      int custno;
      char type;
   };
   struct custrec {
      char name[NAMESIZE];
      char addr[ADDRSIZE];
   };
   typedef double payrec;

   struct hdr header;
   struct custrec customer;
   payrec payment;

   custf = fopen("tso:custfile", "rb");
   if (!custf) exit(1);

   for (;;)
      {
            /* Read customer number and record type.         */
         afreadh(&header, sizeof(header), 1, custf);
         if (feof(custf) || ferror(custf)) break;
         if (header.type == 'C'){        /* a customer record */
                  /* Read rest of customer record. */
            afread(&customer, sizeof(customer), 1, custf);

            if (feof(custf) || ferror(custf)) break;
              printf("Customer record %d read:\n"
                     "Name: %s\nAddress: %s\n", header.custno,
                     customer.name, customer.addr);
            }
         else if (header.type == 'P'){    /* a payment record */
            afread(&payment, sizeof(payment), 1, custf);
            if (feof(custf) || ferror(custf)) break;
            printf("Payment record for customer %d read:\n"
                   "Amount: %.2f\n", header.custno, payment);
            }
         else{
            printf("Unknown record type %c, aborting.\n",
                   header.type);
            abort();
            }
      }

      if (ferror(custf)){
            puts("Aborting due to error reading file.");
            abort();
         }
      fclose(custf);
      exit(0);
}
```

afreadh Read Part of a Record
(continued)

RELATED FUNCTIONS

`afread`

SEE ALSO

"Augmented Standard I/O" on page 3-63
"I/O Functions" on page 2-13

afreopen Reopen a File with System-Dependent Options

SYNOPSIS

```
#include <lcio.h>

FILE *afreopen(const char *name, const char *mode, FILE *oldf,
               const char *am, const char *amparms);
```

DESCRIPTION

The **afreopen** function closes the stream associated with the **FILE** object addressed by **oldf** and then reopens it using the filename, open mode, access method, and amparms specified by the remaining arguments. The **oldf** pointer can also identify a stream that has been closed, in which case, only the open portion of **afreopen** is performed.

The **name** argument is the external name (sometimes called a *pathname*) of the file to be opened. Its form is operating-system-dependent. See "General filename specification" on page 3-28 for syntax details. Note that the **name** to be opened may be different from the filename currently associated with the **oldf** argument.

The **mode** argument is a string defining how the file will be used. The **mode** string contains one to three characters with the following syntax:

```
r  | [b]
w  | [+] | [k]
a
```

r indicates that the file will be read.

w indicates that the file will be written and the previous contents discarded.

a indicates that data will be added to the end of the file. (If **a** is specified, the file is automatically positioned to the end of the file immediately before data are physically written.) For a keyed file, new data can be added anywhere in the file, but existing records cannot be changed.

+ indicates that the file will be both read and written if it is present in the **mode** string.

b requests access as a binary stream, and **k** requests access as a keyed stream. If neither **b** nor **k** is specified, text access is assumed.

Refer to "Open modes" on page 3-33 for more details.

The **am** argument is a string naming an access method. Remember to enclose the method in quotation marks. The following specifications are permitted:

``'' enables the library to select the access method.

term applies only to terminal files.

seq is primarily oriented towards sequential access. It can be specified for terminal files, in which case the **term** access method is automatically substituted.

rel is oriented primarily toward access by relative character number. The **rel** access method can be used only when the open mode specifies binary access.

afreopen Reopen a File with System-Dependent Options
(continued)

kvs is oriented toward access to data by physical or logical keys. The **kvs** access method can be used only with VSAM files and only when the mode specifies keyed access.

fd provides access to OpenEdition HFS files.

If the file cannot be handled with the access method you specify, the **afreopen** operation fails.

See "Library access method selection" on page 3-35 for more information on access method selection.

The **amparms** argument is a string specifying access method parameters, which are system- and access-method dependent file processing options. See "Access method parameters" on page 3-36 for a complete discussion of amparms.

RETURN VALUE

If **afreopen** is successful, the value of **oldf** is returned. The **FILE** object addressed by **oldf** is now associated with the file specified by **name**.

If **afreopen** is unsuccessful, a **NULL FILE** pointer is returned. Further use of **oldf** after an unsuccessful **afreopen** is not permitted.

EXAMPLE

```
#include <lcio.h>

   /* Reopen stdin to the terminal or the DDname */
   /* INPUT. Use "/*" to indicate end of file.   */
afreopen("tso:*INPUT", "r", stdin, "", "eof=/*");
```

RELATED FUNCTIONS

afopen, **freopen**

SEE ALSO

"Opening Files" on page 3-27

"Access method parameters" on page 3-36

"I/O Functions" on page 2-13

afwrite Write a Record

SYNOPSIS

```
#include <lcio.h>

size_t afwrite(const void *ptr, size_t size,
               size_t count, FILE *f);
```

DESCRIPTION

`afwrite` writes items to the stream associated with the **`FILE`** object addressed by **`f`** and then forces a record break. **`size`** defines the size of each item, **`count`** defines the number of items to be written, and **`ptr`** addresses the area containing the items. If all the items do not fit into the current record, a diagnostic message is generated, and the file's error flag is set.

Calls to **`afwrite`** to write items of type **`typeval`** commonly have this form:

```
typeval buf[count];
afwrite(buf, sizeof(typeval), count, f);
```

`afwrite` is supported only for binary streams. See "Augmented Standard I/O" on page 3-63 for more information.

RETURN VALUE

`afwrite` returns the number of items written. If there are too many items, only those that fit are written.

CAUTION

When used on a file with relative attributes, **`afwrite`** behaves exactly like **`fwrite`** because such a file is processed as a continuous stream of characters without record boundaries. To process a file with relative attributes on a record-by-record basis, you must open it with **`afopen`** and specify the **`''seq''`** access method.

If you call **`afwrite`** with a size or count of 0, nothing is written. You must use the **`afwrite0`** function to write a zero-length record.

DIAGNOSTICS

`afwrite` never writes more than a single record; it is an error if there is no room in the current record for all items.

The return value from **`afwrite`** does not indicate whether the call is completely successful. You can use the **`ferror`** function to determine whether an error occurs.

afwrite Write a Record
(continued)

EXAMPLE

This program writes out the same data three different ways using **fputs**, **fwrite**, and **afwrite**. This example illustrates the different ways that these functions handle new lines and record boundaries:

```
#include <lcio.h>
#include <stdlib.h>

main()
{
   FILE *f1, *f2, *f3;
   char *strings[] = {
      "a\nb", "a\nb\nc", "a\nb\nc\nd", "a\nb\nc\nd\ne",
      "a\nb\nc\nd\ne\nf" };
   int i;
      /* Open for text when we use fputs.                     */
   f1 = afopen("cms:fputs output", "w", "", "recfm=v,reclen=20");
   f2 = afopen("cms:fwrite output", "wb", "", "recfm=v,reclen=20");
      /* Open for binary when we use fwrite or afwrite.       */
   f3 = afopen("cms:afwrite output", "wb", "", "recfm=v,reclen=20");


   if (!f1 || !f2 || !f3){
         puts("File(s) failed to open.");
         exit(EXIT_FAILURE);
   }
   for (i = 0; i < sizeof(strings)/sizeof(strings[0]); ++i){
         fputs(strings[i], f1);
         fwrite(strings[i], strlen(strings[i]), 1, f2);
         afwrite(strings[i], strlen(strings[i]), 1, f3);
   }
   fclose(f1); fclose(f2); fclose(f3);
   puts("Compare output files: FPUTS OUTPUT, FWRITE OUTPUT and "
         "AFWRITE OUTPUT.");
   exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

afwrite0, **afwriteh**, **kinsert**, **kreplace**, **fputs**

SEE ALSO

"Augmented Standard I/O" on page 3-63

"I/O Functions" on page 2-13

afwrite0 Write a Record (possibly length 0)

SYNOPSIS

```
#include <lcio.h>

int afwrite0(const void *ptr, size_t size, size_t count, FILE *f);
```

DESCRIPTION

`afwrite0` writes items from the stream associated with the **`FILE`** object addressed by **`f`** and then forces a record break. **`size`** defines the size of each item, **`count`** defines the maximum number of items to be written, and **`ptr`** addresses the area containing the items. If all the items do not fit into the current record, a diagnostic message is generated and the file's error flag is set. A call with a **`count`** of zero causes a zero-length record to be written, if the output file supports it.

Calls to **`afwrite0`** to write items of type **`typeval`** commonly have the form

```
typeval buf[count];
numwrit = afwrite0(buf, sizeof(typeval), count, f);
```

`afwrite0` is supported only for binary streams. See "Augmented Standard I/O" on page 3-63 for more information on **`afwrite0`**.

Note: **`afwrite0`** differs from **`afwrite`** only in the type of the return value and the treatment of a zero **`count`**.

RETURN VALUE

`afwrite0` returns the number of items written. If an error occurs, a negative value is returned.

CAUTION

When used on a file with relative attributes, **`afwrite0`** behaves exactly like **`fwrite`** because these files are processed as continuous streams of characters without record boundaries. To process a file with relative attributes on a record-by-record basis, you must open it with **`afopen`** and specify the **`''seq''`** access method.

DIAGNOSTICS

`afwrite0` never writes more than a single record; an error occurs if there is no room in the current record for all items.

EXAMPLE

See the example for **`afread0`**.

RELATED FUNCTIONS

`afwrite`

SEE ALSO

"Augmented Standard I/O" on page 3-63

"I/O Functions" on page 2-13

afwriteh Write Part of a Record

SYNOPSIS

```
#include <lcio.h>

size_t afwriteh(const void *ptr, size_t size,
                size_t count, FILE *f);
```

DESCRIPTION

`afwriteh` writes items to the stream associated with the `FILE` object that `f` addresses. `size` defines the size of each item, `count` defines the number of items to be written, and `ptr` addresses the area containing the items. If all the items do not fit into the current record, a diagnostic message is generated and the file's error flag is set.

Calls to `afwriteh` to write items of type `typeval` commonly have this form:

```
typeval buf[count];
afwriteh(buf, sizeof(typeval), count, f);
```

You can only use `afwriteh` with a binary stream. See "Augmented Standard I/O" on page 3-63 for more information.

RETURN VALUE

`afwriteh` returns the number of items written. If there are too many items, only those that fit are written.

CAUTION

When used on a file with relative attributes, `afwriteh` behaves exactly like `fwrite` because such a file is processed as a continuous stream of characters without record boundaries. To process a file with relative attributes on a record-by-record basis, you must open it with `afopen` and specify the `''seq''` access method.

DIAGNOSTICS

`afwriteh` never writes more than a single record; an error occurs if there is no room in the current record for all items.

The return value from `afwriteh` does not indicate whether the call is completely successful. You can use the `ferror` function to determine whether an error occurs.

afwriteh Write Part of a Record
(continued)

EXAMPLE

This example writes a customer record to be read by the **afreadh** example:

```
#include <lcio.h>
#include <stdlib.h>

#define NAMESIZE 30
#define ADDRSIZE 80

main()
{
   FILE *custf;
   struct hdr {
      int custno;
      char type;
   };
   struct custrec {
      char name[NAMESIZE];
      char addr[ADDRSIZE];
   };
   typedef double payrec;

   int customers[] = {
      1001, 1002, 1003, 1004};
   struct custrec custinfo[] = {
      { "Paul Barnes", "256 Oak Street, Cary, NC" },
      { "Janice Palmer", "1500 Pine Avenue, Austin, TX" },
      { "Frank Smith", "92 Maple Boulevard, Concord, NH" },
      { "Carlotta Perez", "634 First Street, Los Angeles, CA" }
    };
   payrec payment[] = { 54.40, 234, 16.81, 523};

   struct hdr cust_hdr;
   int i;

   custf = fopen("tso:custfile", "wb");
   if (!custf)
      exit(1);

      /* Write out one customer record and one payment record */
      /* for each customer.                                    */
   for (i = 0; i < sizeof(customers)/sizeof(customers[0]); ++i){
      cust_hdr.custno = customers[i];
      cust_hdr.type = 'C';
      afwriteh(&cust_hdr, sizeof(cust_hdr), 1, custf);
      if (ferror(custf)) exit(1);
      afwrite(&custinfo[i], sizeof(struct custrec), 1, custf);
      if (ferror(custf)) exit(1);
      cust_hdr.type = 'P';
      afwriteh(&cust_hdr, sizeof(cust_hdr), 1, custf);
      if (ferror(custf)) exit(1);
      afwrite(&payment[i], sizeof(payrec), 1, custf);
```

afwriteh Write Part of a Record
(continued)

```
         if (ferror(custf)) exit(1);
      }
      fclose(custf);
      printf("%d customer records and %d payment records written.\n",
             sizeof(customers)/sizeof(customers[0]),
             sizeof(customers)/sizeof(customers[0]));
      exit(0);
   }
```

RELATED FUNCTIONS

`afwrite`

SEE ALSO

"Augmented Standard I/O" on page 3-63

"I/O Functions" on page 2-13

alarm, alarmd Request a Signal after a Real-Time Interval

SYNOPSIS

```
#include <lclib.h>

unsigned int alarm(unsigned int sec);
double alarmd(double sec);
```

The synopsis for the POSIX implementation is as follows:

```
#include <sys/types.h>
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

You may use either set of header files in your program.

DESCRIPTION

alarm and **alarmd** request that a SIGALRM signal be generated after the number of seconds specified by its argument. Any previous call to **alarm** or **alarmd** is canceled. If the argument to either of the alarm functions is 0, any previous **alarm** or **alarmd** request is canceled, but no SIGALRM signal is generated. An argument greater than a day (86400 seconds) is treated as a day.

The SIGALRM signal is asynchronous, so it is discovered only when a function is called or returns. For this reason, as well as because of competition from other users, the signal may take slightly longer than the specified amount of time to be generated.

alarmd performs the same actions as **alarm** but permits the amount of time to be specified with greater accuracy. The accuracy of timing depends on the operating system and CPU model.

RETURN VALUE

The alarm functions return the number of seconds remaining from any previous interval. (**alarm** rounds up to an integer of seconds.) If no interval is currently active, 0 is returned.

CAUTION

Under VM/SP and VM/HPO CMS, you must use the CP command SET TIMER REAL for proper functioning of **alarm** and **alarmd**. If SET TIMER REAL is not in effect, a diagnostic message is produced and a SIGALRM signal is generated immediately.

If SIGALRM is handled by OpenEdition, **alarmd** is not available.

PORTABILITY

alarmd is not portable.

alarm, alarmd
(continued)

Request a Signal after a Real-Time Interval

EXAMPLE

This example counts the number of divisions you can do in 5 seconds:

```
#include <lclib.h>
#include <signal.h>
#include <setjmp.h>
#include <lcjmp.h>
#include <stdio.h>

void timeout(int signum);
jmp_buf jbuf;
volatile int i;
int jcode;

main()
{
      /* Establish SIGALRM handling.                    */
   onjmp(jbuf, jcode, done);
   signal(SIGALRM, &timeout);

      /* Perform calculations.                          */
   alarm(5);
   for (i = 1; ; i++) {
      i/=1;
      sigchk();
   }
   done:
       printf("%d divisions executed in 5 seconds.\n", i);
   return;

}

   /* SIGALRM handler gets out of loop.                 */
void timeout(int signum)
{
  longjmp(jbuf, 1);
}
```

SEE ALSO

"SIGALRM" on page 5-23

"Signal-Handling Functions" on page 2-17

aopen Open a UNIX-Style File for I/O with Amparms

SYNOPSIS

```
#include <fcntl.h>

int aopen(const char *name, int flags, const char *amparms);
```

DESCRIPTION

aopen is a variant of **open**. It enables you to open a file for UNIX style I/O while specifying access method parameters (amparms) to request 370-dependent options.

The **name** argument is the external name (sometimes called *pathname*) of the file to be opened. Its form depends on the operating system. If **name** is an HFS file, the file is not opened directly. The file is opened through standard I/O as though it were an MVS file because **aopen** is used to specify amparms. OpenEdition does not support amparms. This should be a rare occurrence because there is little reason to call **aopen** for an HFS file.

The **flags** argument is a bit string formed by ORing option bits. The bits are defined symbolically, and the header file **<fcntl.h>** should be included to obtain their definitions. The flags and their meanings are as follows:

O_RDONLY	specifies to open for reading only.
O_WRONLY	specifies to open for writing only.
O_RDWR	specifies to open for both reading and writing.
O_APPEND	specifies to seek to end of file before each write.
O_CREAT	specifies to create a new file if it does not exist.
O_TRUNC	specifies to discard old data from existing file.
O_EXCL	specifies to not accept an existing file.
O_TEXT	specifies to open for text access.
O_BINARY	specifies to open for binary access.
O_NONBLOCK	specifies the use of nonblocking I/O. This option is meaningful only for OpenEdition HFS files.
O_NOCTTY	specifies that the file is not to be treated as a controlling terminal. This option is meaningful only for OpenEdition HFS files.

You should only set one of the following options: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. **O_EXCL** is ignored if **O_CREAT** is not also set. If neither **O_TEXT** nor **O_BINARY** is specified, **O_BINARY** is assumed unless the file to be opened is the terminal.

The **amparms** argument is a string specifying access method parameters, which are system- and access-method-dependent file processing options.

See "Open modes" on page 3-33 for more information on details of the filename, open mode, and amparms specifications.

RETURN VALUE

aopen returns the file number of the file that was opened. If it fails, **aopen** returns −1.

aopen Open a UNIX-Style File for I/O with Amparms
(continued)

IMPLEMENTATION

You can use files opened with **aopen** and files opened with **open** interchangeably. The **name** and **mode** arguments to **aopen** have the same meanings and formats as the corresponding **open** arguments. **aopen(name, mode, \`\`'')** is equivalent to **open(name, mode)**.

EXAMPLE

```
#include <fcntl.h>
#include <stdlib.h>

main()
{
  int cardfd;
  char ending = '.';

     /*  Open a card-image file.                              */
  cardfd = aopen("tso:cards.out", O_WRONLY | O_CREAT | O_TRUNC,
                 "recfm=f,reclen=80");

  if (cardfd < 0) exit(EXIT_FAILURE);
  lseek(cardfd, 7999, SEEK_SET);
     /* Write a '.' in position 7999. Previous positions     */
     /*  will be filled with nulls.                           */
  write(&ending, cardfd, 1);
  close(cardfd);
  exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

afopen

SEE ALSO

"Opening Files" on page 3-27

"UNIX style I/O" on page 3-17

"I/O Functions" on page 2-13

asctime Convert Time Structure to Character String

SYNOPSIS

```
#include <time.h>

char *asctime(const struct tm *timeinfo);
```

DESCRIPTION

asctime converts a broken-down time value (as stored in a **tm** structure) to a printable character string and returns the address of the first character of the string.

The string has the form "**wkd mon dd hh:mm:ss yyyy\n**", for example "**Thu Oct 10 16:49:07 1985\n**". The length of the string is always 25. The day of the month is padded with blanks on the left to two characters, if necessary (for example, Oct 09). The hours, minutes, and seconds are padded with 0s.

RETURN VALUE

asctime returns a pointer to the formatted date and time.

CAUTION

The pointer returned by **asctime** may reference **static** storage, which may be overwritten by the next call to **asctime** or **ctime**.

EXAMPLE

```
#include <stdio.h>
#include <time.h>

main ()
{
   time_t timeval;
   struct tm *now;

   time(&timeval);
   now = gmtime(&timeval);   /* Get current GMT time */
   printf("The current GMT time and date are: %s",
          asctime(now));
}
```

RELATED FUNCTIONS

ctime, **strftime**

SEE ALSO

"Timing Functions" on page 2-11

asin Compute the Trigonometric Arc Sine

SYNOPSIS

```
#include <math.h>

double asin(double x);
```

DESCRIPTION

asin computes the trigonometric arc sine of the value **x**. The arc sine function is the inverse of the sine function and is expressed by the following relation:

$$r = \sin^{-1}(x)$$

x is in the closed interval [−1.0,1.0].

RETURN VALUE

asin returns the principal value of the arc sine of the argument **x**, provided that this value is defined and computable. The return value is a double precision floating-point number in the closed interval [$-\pi/2,\pi/2$] radians.

DIAGNOSTICS

asin returns 0.0 if the value of its argument is larger than 1.0 or smaller than −1.0. The run-time library writes an error message to the standard error file (**stderr**) in this case.

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

EXAMPLE

This example computes pi as 4 times the arc sine of the square root of 0.5:

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

main()
{
   double pival, value;
   value = .500;

   pival = 4 * asin(sqrt(value));
   printf("4 * asin(sqrt(%f)) = %f\n",value,pival);
}
```

RELATED FUNCTIONS

acos, **atan**, **_matherr**

SEE ALSO

"Mathematical Functions" on page 2-6

assert Put Diagnostics into Programs

SYNOPSIS

```
#include <assert.h>

void assert(int expr);
```

DESCRIPTION

assert puts diagnostics into programs. **expr** is an expression. If the macro name **NDEBUG** is undefined at the point in the source file where **<assert.h>** is included, then **assert** expands to a statement that tests the expression. If the expression is false (that is, compares equal to 0), **assert** writes the text of the argument, the source filename as defined by **__FILE__**, and the source line number as defined by **__LINE__** to **stderr** (the standard error file). It then calls **abort**.

If **NDEBUG** is defined at the point in the program where **<assert.h>** is included, then **assert** expands to **(void *)0**.

The diagnostic is in the format of a normal library diagnostic, with the following text:

```
Assertion failed: expr
Interrupted while: Executing line number of source-file
```

expr is the expression, *number* is the current value of **__LINE__**, and *source-file* is the current value of **__FILE__**.

RETURN VALUE

assert has no return value.

CAUTION

assert generates a library diagnostic. Because the **quiet** function suppresses library diagnostics, it also suppresses **assert** diagnostics.

You should suppress assertions by defining **NDEBUG**, not by calling **quiet**; **quiet** will have no effect on the run-time overhead of verifying the assertions.

USAGE NOTES

The macro **NDEBUG** is automatically defined by the compiler, unless you use the **DEBug** option. Suppress this automatic definition by using the **UNdef** option or by using **NDEBUG** in a **#undef** preprocessor directive. If you use the **DEBug** option, then **NDEBUG** is not automatically defined.

assert Put Diagnostics into Programs
(continued)

EXAMPLE

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

double arcsin(double x)  {
   assert(x <= 1.0 && x >= -1.0);
   return asin(x);
}

main()
{
   double num, svalue;

   puts("Enter a number.");
   scanf("%f",&num);
   svalue = arcsin(num);
   printf("The arcsin of the number is %f \n", svalue);
}
```

RELATED FUNCTIONS

quiet

SEE ALSO

"Diagnostic Control Functions" on page 2-11

atan Compute the Trigonometric Arc Tangent

SYNOPSIS

```
#include <math.h>

double atan(double x);
```

DESCRIPTION

atan computes the trigonometric arc tangent of its argument **x**. The arc tangent is the inverse of the tangent function and is expressed by the following relation:

$$r = \tan^{-1}(x)$$

RETURN VALUE

atan returns the principal value of the arc tangent of the argument **x**, provided that this value is defined and computable. The return value is a double-precision, floating-point number in the open interval $(-\pi/2, \pi/2)$ radians.

DIAGNOSTICS

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

EXAMPLE

This example computes pi as 4 times the arc tangent of 1:

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

main()
{
   double pival, value;
   value = 1.000;

   pival = 4 * atan(value);
   printf("4 * atan(%f) = %f\n", value, pival);
}
```

RELATED FUNCTIONS

acos, **asin**, **atan2**, **_matherr**

SEE ALSO

“Mathematical Functions” on page 2-6

atan2 Compute the Trigonometric Arc Tangent of a Quotient

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double atan2(double y, double x);
```

DESCRIPTION

`atan2` computes the angle defined by the positive x axis and a line through the point (`x,y`) to the point (0, 0). The signs of both values `x` and `y` are used to determine the quadrant of the result in a Cartesian system. The result is the inverse trigonometric tangent of `y/x` if `x` is not 0.

RETURN VALUE

Provided that it is defined and computable, the return value is the angular position of the point `x,y`. The return value is a double-precision, floating-point number expressed in radians and lies in the half-open interval $(-\pi,\pi]$. For input values (0.0, `y`), the return value will be either $\pi/2$ or $-\pi/2$ if `y` does not equal 0.

DIAGNOSTICS

If both `x` and `y` are 0, an error message is written to `stderr` and the function returns 0.0.

If an error occurs in `atan2`, the `_matherr` routine is called. You can supply your own version of `_matherr` to suppress the diagnostic message or modify the value returned.

EXAMPLE

This example converts rectangular coordinates to polar coordinates:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double x, y;            /* rectangular coordinates (x,y) */
   double r, angle;        /* polar coordinates (r,angle)   */

   puts("Enter the rectangular coordinates please: ");
   scanf("%lf %lf", &x, &y );

   r = sqrt((x*x) + (y*y));

      /* Compute polar coordinates (radians).               */
   angle = atan2(y , x);

      /* Convert radians to degrees.                        */
   angle = (180.0 * angle)/M_PI;
```

atan2 Compute the Trigonometric Arc Tangent of a Quotient
(continued)

```
      printf("rect coords(%f,%f) -> polar coords(%f,%f)n",
             x,y,r,angle);
   }
```

RELATED FUNCTIONS

atan, **_matherr**

SEE ALSO

"Mathematical Functions" on page 2-6

atexit Register Program Cleanup Function

SYNOPSIS

```
#include <stdlib.h>

int atexit(void (*func)());
```

DESCRIPTION

atexit defines a function to be called during program termination, either as the result of a call to **exit** or a return from the **main** function. The **func** argument should be a function with no arguments returning **void**. Usually, **atexit** routines are used to free resources allocated by the program that are not freed automatically by the library, such as memory allocated by direct use of GETMAIN or DMSFREE. You can call **atexit** any number of times, and you can register the same routine more than once, in which case it is called once for each registration.

atexit cleanup routines are called in the opposite order of their registration. They are called before any coprocesses are terminated or open files are closed, but after termination of any active functions. (Thus, a cleanup routine cannot cause execution to resume by issuing **longjmp**.) A cleanup routine can call **exit**, which has no effect other than possibly to change the exit code returned to the operating system. In this case, no cleanup routine that was previously called is called again.

It is not possible to deregister a function after it is registered. However, when a load module containing a registered cleanup routine is unloaded using **unloadm**, the cleanup routine is deregistered automatically.

It is not possible to define additional cleanup routines after program termination starts.

RETURN VALUE

atexit returns 0 if it is successful, or a non-zero value if it is unsuccessful.

The ISO/ANSI Standard for C permits an implementation to enforce an upper limit of 32 registered functions and leaves the results undefined if a registered function calls **exit**.

USAGE NOTES

Both **atexit** and **blkjmp** enable you to intercept calls to the **exit** function. You should consider the following when deciding which function to use:

- □ **atexit** is portable and **blkjmp** is not.
- □ **blkjmp** cannot intercept return from the **main** function.

atexit Register Program Cleanup Function
(continued)

- **`atexit`** cannot prevent completion of **`exit`** but **`blkjmp`** can.
- **`blkjmp`** is used mainly for intercepting termination of an active function, either by **`exit`** or by **`longjmp`** to a calling routine. **`atexit`** has no effect on **`longjmp`** calls but can be used without concern for active functions.

EXAMPLE

This example allocates a buffer using the CMS DMSFREE facility and defines an **`atexit`** routine to release it because CMS leaves it allocated forever.

```
#include <stdlib.h>
#include <stdio.h>
#include <dmsfree.h>

static void cleanup();
extern char *buffer;
static unsigned bufsize;

void getbuf(unsigned int size)
{
      /* Allocate buffer; check for error.                  */
   if (DMSFREE((size+7)/8, &buffer, FREE_DEF, ERR_RET)){
      puts("Unable to allocate buffer.");
      exit(16);
   }

      /* Save buffer size.                                  */
   bufsize = (size+7)/8;

      /* If exit not defined, cleanup without it and quit.  */
   if (atexit(&cleanup) != 0) {
      cleanup();
      exit(16);
   }
}

static void cleanup()
{
      /* Return buffer at end of execution.                 */
   DMSFRET(bufsize, buffer, MSG_YES, ERR_ABN);
}
```

RELATED FUNCTIONS

`atcoexit`, **`blkjmp`**, **`exit`**

SEE ALSO

"Program Control Functions" on page 2-9

atof Convert a String to Floating Point

SYNOPSIS

```
#include <stdlib.h>

double atof(const char *p);
```

DESCRIPTION

atof converts the character string **p** to a double-precision, floating-point number after skipping any leading white space (such as blanks, tabs, and new-line characters). The conversion stops at the first unrecognized character.

The argument string may contain a decimal point and may be followed by an **e** or an **E** and a signed integer exponent. A leading minus sign indicates a negative number. White space is not allowed between the minus sign and the number or between the number and the exponent.

RETURN VALUE

atof returns a value of type **double**. If no initial segment of the string is a valid number, the return value is 0.

DIAGNOSTICS

No indication is returned to the program to specify whether the string contains a valid number, so you should validate the string before calling **atof**.

If the floating-point value is outside the range of valid 370 floating-point numbers, ± **HUGE_VAL** 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 <stdio.h>
#include <string.h>
#include <ctype.h>

main() {
   char input[80];
   int int_format;
   long int_value;
   double dbl_value;

   puts("Enter a valid C numeric constant (without any suffixes)");
   gets(input);
      /* If there's a decimal point, it's a double.   */
   if (strchr(input, '.')) int_format = 0;
      /* If it starts 0x, it's an integer.            */
   else if((input[0] == '0' && tolower(input[1]) == 'x') ||
           ((input[0] == '+' || input[0] == '-') &&
            input[1] == '0' && tolower(input[2]) == 'x'))
      int_format = 1;
      /* If it has an E and isn't hex, it's a double. */
   else if (strpbrk(input, "eE")) int_format = 0;
     /* Doubles must have either "." or "e".          */
```

atof Convert a String to Floating Point
(continued)

```
      else int_format = 1;

         /* Convert to integer (errors ignored).         */
      if (int_format) {
         int_value = strtol(input, NULL, 0);
         printf("Your input appears to be the integer %d\n",
               int_value);
      }
         /* Convert to double (errors ignored).          */
      else {
         dbl_value = atof(input);
         printf("Your input appears to be the double %.16g\n",
               dbl_value);
      }
      exit(0);
   }
```

RELATED FUNCTIONS

`strtod`

SEE ALSO

"String Utility Functions" on page 2-3

atoi Convert a String to Integer

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

int atoi(const char *str);
```

DESCRIPTION

atoi converts the character string **str** to an integer. The string should consist of an optional plus or minus sign, followed by one or more decimal digits. Initial white-space characters are ignored.

RETURN VALUE

atoi returns the integer value represented by the character string up to the first unrecognized character. If no initial segment of the string is a valid integer, the return value is 0.

CAUTION

No indication of overflow or other error is returned, so you should validate the string before calling **atoi**.

DIAGNOSTICS

No indication is returned to the program to specify whether the string contains a valid integer.

If the correct value is too large to be stored in a 370 **long**, either **LONG_MAX** ($2^{31}-1$) or **LONG_MIN** (-2^{31}) is returned, depending on the sign of the value.

IMPLEMENTATION

atoi(x) is implemented as **(int)atol(x)**.

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXLINE 25

main()
{
   int value;
   char *input;
   char line[MAXLINE];

   puts("Enter a number: ");
   input = gets(line);
```

atoi Convert a String to Integer
(continued)

```
         /* If string contains only digits, white space, or +/-,  */
         /*  convert. Note that the input may still not represent */
         /*  a valid integer.  Consider +-14+5.                    **/
      if (strspn(input, " +-0123456789") == strlen(input)){
         value = atoi(input);
         printf(" The integer equivalent of given string is %d\n",
               value);
      }
      else
         printf("Invalid count value: %s\n", input);
   }
```

RELATED FUNCTIONS

strtol

SEE ALSO

"String Utility Functions" on page 2-3

atol Convert a String to Long

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

long int atol(const char *str);
```

DESCRIPTION

atol converts the character string **str** to a **long** integer. The string should consist of an optional plus or minus sign, followed by one or more decimal digits. Initial white-space characters are ignored.

RETURN VALUE

atol returns the integer value represented by the character string, up to the first unrecognized character. If no initial segment of the string is a valid integer, the return value is **0L**.

CAUTION

No indication of overflow or other error is returned, so you should validate the string before calling **atol**.

DIAGNOSTICS

No indication is returned to the program to specify whether the string contains a valid integer.

If the correct value is too large to be stored in a 370 **long**, either **LONG_MAX** ($2^{31}-1$) or **LONG_MIN** (-2^{31}) is returned, depending on the sign of the value.

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAXLINE 25

main()
{
  long value;
  char *input;
  char line[MAXLINE];

  puts("Enter a number: ");
  input = gets(line);
```

atol Convert a String to Long
(continued)

```
         /* If string contains only digits, white space, or +/-,  */
         /*  convert. Note that the input may still not represent */
         /*  a valid integer. Consider +-14+5.                    */
      if (strspn(input, " +-0123456789") == strlen(input)){
         value = atol(input);
         printf("The long integer equivalent of given string is %ld\n",
         value);
      }
      else
        printf("Invalid count value: %s\n", input);
   }
```

RELATED FUNCTIONS

strtol

SEE ALSO

"String Utility Functions" on page 2-3

blkjmp Intercept Nonlocal gotos

SYNOPSIS

```
#include <lcjmp.h>

int blkjmp(jmp_buf env);
```

DESCRIPTION

blkjmp requests interception of calls to **longjmp** that could terminate the calling function. When you call **blkjmp**, it always returns 0. If a call to **longjmp** is later intercepted, the call to **blkjmp** is resumed and it then returns the integer argument that was passed to **longjmp**. The **env** variable is modified to indicate the target of the intercepted **longjmp** so that it can be reissued by the intercepting routine.

After a call to **longjmp** is intercepted, **blkjmp** must be reissued if continued interception is wanted.

Because **exit** is implemented as a **longjmp** to the caller of **main**, you can use **blkjmp** to intercept program exit.

RETURN VALUE

blkjmp normally returns 0; it returns a non-zero value if a call to **longjmp** has been intercepted (in which case, **blkjmp** returns the value of the second argument passed to **longjmp**).

CAUTION

Variables of storage class **auto** and **register** whose values are changed between the **blkjmp** and **longjmp** calls have indeterminate values on return to **blkjmp**.

EXAMPLE

This example demonstrates how **blkjmp** can be used to enable a function to release resources, even if terminated by a call to **longjmp** in a function it calls:

```
#include <stdio.h>
#include <lcjmp.h>
#include <stdlib.h>

jmp_buf env;

static void get_resource(void), use_resource(void);

int main()
{
   int code;
```

blkjmp Intercept Nonlocal gotos
(continued)

```
   if (code = setjmp(env)) goto escape;
   get_resource();
   puts("get_resource returned normally.");
   exit(0);
      escape:
         printf("Executing escape routine for error %d\n", code);
         exit(code);
}

static void get_resource(void)
{
   int code;
   jmp_buf my_env;

      /* Allocate resource here.       */
   if (code = blkjmp(my_env)) goto release;
   puts("Resources allocated.");
      /* Free resource here.           */
   use_resource();
   puts("use_resource returned normally, "
        "get_resource is freeing resources.");
   return;
         /* Free resource here.        */
      release:
         printf("use_resource indicated error %d\n", code);
         puts("Resources now freed, proceeding with longjmp.");
         longjmp(my_env, code);
}

static void use_resource(void)
{
   puts("Entering use_resource");

      /* Attempt to use resource here. */
   puts("Error 3 detected, calling longjmp.");
   longjmp(env, 3);
   puts("This statement will not be executed.");
}
```

RELATED FUNCTIONS

longjmp, **setjmp**, **siglongjmp**, **sigsetjmp**

SEE ALSO

"Program Control Functions" on page 2-9

bsearch Perform a Binary Search

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

void *bsearch(const void *srch, const void *blk,
              size_t n, size_t size,
              int (*cmp)(const void *, const void *));
```

DESCRIPTION

bsearch scans a sorted memory block addressed by **blk** for a match with a search value addressed by **srch**. The **blk** function is a pointer to the first element of the memory block to be scanned. **n** designates the number of elements in the block, and **size** is the size of each element in bytes.

bsearch calls a user-provided comparison function, **cmp**, and passes **cmp** pointers to the two objects being compared.

cmp must return the following values:

- a negative integer, if the first of the two objects is less than the second
- a positive integer, if the first object is greater than the second
- 0, if the two objects are equal.

The description of **qsort** discusses comparison function requirements in more detail.

RETURN VALUE

bsearch returns a pointer to the element that matches the search value. If no match can be found, **NULL** is returned.

EXAMPLE

See the example for **qsort**.

RELATED FUNCTIONS

qsort

SEE ALSO

"General Utility Functions" on page 2-9

btrace Generate a Traceback

SYNOPSIS

```
#include <lclib.h>

void btrace(void (*fp)(const char *));
```

DESCRIPTION

btrace generates a library traceback similar to the traceback generated when an ABEND occurs. **btrace** is useful for debugging or diagnosing error conditions.

fp may be **NULL** or a pointer to a function. If it is **NULL**, then **btrace** writes the traceback to **stderr** (the standard error file). If **fp** is not **NULL**, then **btrace** calls the function for each line of the traceback. One traceback line, in the form of a string, is passed to the function. (The traceback line does not terminate in a newline character.)

RETURN VALUE

btrace has no return value.

EXAMPLE

This example defines a function to write traceback lines to a file:

```
#include <lclib.h>
#include <stdio.h>

static void btrace_out(const char *line)
{
   static FILE *tbf;

      /* If first call, open traceback file.      */
   if (tbf == NULL) {
      tbf = fopen("TRACBACK", "w");
      if (tbf == NULL) exit(12);
      }

      /* Write one line of traceback information. */
   fputs(line, tbf);
   putc('\n', tbf);
}

   /* Define a function to send a message         */
   /*  to stderr and then call btrace().          */
void genbtrac(char *msg)
{
   fputs(msg, stderr);
   btrace(&btrace_out);
   fputs("Traceback generated.\n", stderr);
   exit(12);
}
```

btrace Generate a Traceback
(continued)

SEE ALSO

"Diagnostic Control Functions" on page 2-11

calloc Allocate and Clear Memory

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

void *calloc(size_t n, size_t size);
```

DESCRIPTION

`calloc` allocates a block of dynamic memory to contain **`n`** elements of the size specified by **`size`**. The block is cleared to binary 0s before return.

RETURN VALUE

`calloc` 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 or if you request 0 bytes, **`NULL`** is returned.

IMPLEMENTATION

The size of the block allocated is at least **`n * size`**.

Under an XA or ESA operating system, memory allocated by **`calloc`** can reside above the 16-megabyte line for programs that run in 31-bit addressing mode.

See **`malloc`** for further implementation information.

EXAMPLE

This function allocates a square identity matrix (one in which all elements are zeroes except for those on the diagonal). The number of rows and columns is passed as an argument. If memory cannot be allocated, a NULL pointer is returned.

```
#include <stdio.h>
#include <stdlib.h>

double *identity(int size) {
   double *matrix;
   int i;

   matrix = calloc(sizeof(double), size*size);
   if (matrix == NULL) return(NULL);
   for (i = 0; i < size; ++i)
      matrix[size*i + i] = 1.0;
   return matrix;
}
```

calloc Allocate and Clear Memory
(continued)

RELATED FUNCTIONS

`malloc`

SEE ALSO

"Memory Allocation Functions" on page 2-10

ceil Round Up a Floating-Point Number

SYNOPSIS

```
#include <math.h>

double ceil(double y);
```

DESCRIPTION

ceil rounds up a real number to the next larger integral real number.

RETURN VALUE

ceil accepts a floating-point argument **y** and rounds it up to the next larger integer, expressed as a floating-point number.

IMPLEMENTATION

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

EXAMPLE

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

int moda, modb, df;
double score, rank;

main()
{
   puts("Enter two integers: ");
   scanf("%d %d", &moda, &modb);
   puts("Enter an integer divisor: ");
   scanf("%d",&df);

      /* Add the two numbers, divide by the given divisor, and */
      /*  then round up to the closest integer greater than    */
      /*  the result.                                          */
   score = (moda + modb);
   score /= df;
   rank = ceil(score);

      /* Print this rounded result (its "ceil"ed value).       */
   printf("The ceiling of (%d + %d)/ %d = %f\n",moda,modb,df,rank);
}
```

RELATED FUNCTIONS

floor

SEE ALSO

"Mathematical Functions" on page 2-6

chdir Change Directory

SYNOPSIS

```
#include <unistd.h>

int chdir(const char *pathname);
```

DESCRIPTION

chdir changes the working directory to **pathname**. The **pathname** function must specify the name of a file in the OpenEdition HFS. See "File Naming Conventions" on page 3-56 for information on specifying OpenEdition file names.

RETURN VALUE

chdir returns 0 if it is successful and −1 if it is not successful.

IMPLEMENTATION

When you call **chdir** in an application compiled without the **posix** option, the directory name will be interpreted according to the normal rules for interpretation of filenames. The directory name should include a style prefix if the default style is not **''hfs''**.

EXAMPLE

```
/* This example must be compiled with POSIX to run successfully. */

#include <stdio.h>
#include <unistd.h>

char wrkdir[FILENAME_MAX];

main()
{
      /* Change the working directory to /bin.       */
   if (chdir("/bin") != 0)
      perror("chdir() to /bin failed");
   else {
         /* Determine the current working directory. */
      if (getcwd(wrkdir,sizeof(wrkdir)) == NULL)
         perror("getcwd() error");
      else
         printf("Current working directory is: %s\n",wrkdir);
   }
}
```

RELATED FUNCTIONS

getcwd

chdir Change Directory

(continued)

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

“File Management Functions” on page 2-16

chmod Change Directory or File Mode

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
```

DESCRIPTION

`chmod` changes the mode bits for the directory or file specified by **`pathname`**.
The following symbols are defined in the **`<sys/stat.h>`** include file:

`S_ISUID`	sets the execution user ID. When the specified file is processed through an **`exec`** function, the user id of the file owner becomes the effective user ID of the process.
`S_ISGID`	sets execution group ID. When the specified file is processed through an **`exec`** function, the group ID that owns the file becomes the effective group ID of the process.
`S_ISVTX`	specifies shared text.
`S_IRUSR`	sets file owner permission to read.
`S_IWUSR`	sets file owner permission to write.
`S_IXUSR`	sets file owner permission to execute.
`S_IRWXU`	sets file owner permission to read, write, and execute.
`S_IRGRP`	sets group permission to read.
`S_IWGRP`	sets group permission to write.
`S_IXGRP`	sets group permission to execute.
`S_IRWXG`	sets group permission to read, write, and execute.
`S_IROTH`	sets general permission to read.
`S_IWOTH`	sets general permission to write.
`S_IXOTH`	sets general permission to execute.
`S_IRWXO`	sets general permission to read, write, and execute.

A process can set mode bits if it has superuser authority, or if the user ID is the same as that of the file's owner. The S_ISGID bit in the file's mode bits is cleared if

- □ the calling process does not have superuser authority.
- □ the group ID of the file does not match the effective group ID or any of the process' supplementary group IDs.

RETURN VALUE

`chmod` returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

This example changes a file (named by the function argument) so that it can be executed by any user who can read it.

chmod Change Directory or File Mode
(continued)

```
#include <sys/types.h>
#include <sys/stat.h>

int chexec(const char *name) {
   struct stat stat_data;
   mode_t newmode;
   int rc;

   rc = stat(name, &stat_data);
   if (rc != 0) {
      perror("stat failure");
      return -1;
   }
   newmode = stat_data.st_mode;
   if (newmode & S_IRUSR) newmode |= S_IXUSR;
   if (newmode & S_IRGRP) newmode |= S_IXGRP;
   if (newmode & S_IROTH) newmode |= S_IXOTH;

      /* If the mode bits changed, make them effective. */
   if (newmode != stat_data.st_mode) {
      rc = chmod(name, newmode);
      if (rc != 0) perror("chmod failure");
      return rc;
   }
   return(0);              /* No change was necessary. */
}
```

RELATED FUNCTIONS

chown, **fchmod**

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

“File Management Functions” on page 2-16

clearenv Delete Environment Variables

POSIX

SYNOPSIS

```
#include <env.h>

int clearenv(void);
```

DESCRIPTION

clearenv deletes the environment-variable list for a process. Only program-scope environment variables are affected.

RETURN VALUE

clearenv returns 0 if it is successful and −1 if it is not successful.

CAUTION

A copy of the **environ** pointer may not be valid after a call is made to **clearenv**.

EXAMPLE

The following code fragment illustrates the use of **clearenv**:

```
/* This example requires compilation with the posix option */
/*  to execute successfully.                               */

#include <stdlib.h>
#include <stdio.h>

extern char **environ;

int count_env() {
   int num;

   for (num=0; environ[num] !=NULL; num++);
   return num;
}
.
.
.
printf("There are %d environment variables\n", count_env());

if (clearenv() !=0)
   perror("clearenv() error");
else {
   printf("Now there are %d environment variables\n", count_env());
}
.
.
.
```

clearenv Delete Environment Variables
(continued)

RELATED FUNCTIONS

`getenv`, `setenv`

SEE ALSO

Chapter 4, "Environment Variables" on page 4-1

"System Interface and Environment Variables" on page 2-17

clearerr Clear Error Flag

SYNOPSIS

```
#include <stdio.h>

void clearerr(FILE *f);
```

DESCRIPTION

clearerr clears the error flag and the end-of-file flag for the **FILE** object addressed by **f**.

RETURN VALUE

clearerr has no return value.

IMPLEMENTATION

After the error flag is set for a file, all further I/O to the file fails until the flag is cleared. The standards do not address this subject, and different implementations treat this situation differently. For maximum portability, the error flag should be cleared immediately upon detection of an error if continued use of the file is intended.

In some cases of a severe error, it is impossible to continue to use the file. This situation cannot be detected by **clearerr**. The nonstandard function **clrerr** enables you to test for this situation.

EXAMPLE

```
#include <lcio.h>

int fixerr(FILE *f){            /* Clear the error flag for a file. */
   clearerr(f);
   if (ferror(f)){              /* if error flag still set          */
      printf("Error on %s is permanent. Program aborted.\n", fnm(f));
      abort();
   }
   return 0;                    /* Show error flag cleared.         */
}
```

RELATED FUNCTIONS

clrerr, **ferror**

SEE ALSO

"Error handling" on page 3-62

"I/O Functions" on page 2-13

clock Measure Program Processor Time

SYNOPSIS

```
#include <time.h>

clock_t clock(void);
```

DESCRIPTION

clock returns the amount of processor time used by the program relative to some base point. The amount of processor time used between two calls is indicated by the difference between the values returned by the two calls to **clock** in the same program.

The value returned is of type **clock_t**. The value returned is in fractions of a second, where a value of **CLOCKS_PER_SEC** represents one second of processor time. (**clock_t** and **CLOCKS_PER_SEC** are defined in **<time.h>**.) In this implementation, **clock_t** is defined as a **double** and **CLOCKS_PER_SEC** is 1.0.

RETURN VALUE

clock returns the number of seconds since the base time. If an accurate value cannot be returned, **(clock_t)-1** is returned.

CAUTION

The value returned by **clock** is of relatively low accuracy and may depend on the extent of other system activity. Values returned by **clock** are likely to be inconsistent from one execution of a program to another.

PORTABILITY

For portability's sake, you should always use **CLOCKS_PER_SEC** as a scale factor when using the value returned by **clock**. Also, you should declare variables that contain **clock** values as **clock_t** because many implementations define this type as **long int** or **unsigned long int**.

IMPLEMENTATION

Under MVS, the base point for **clock** is the first call; that is, the first call of **clock** in an MVS program always returns 0.0.

If the program calls the **system** function, processor time subsequently used by invoked programs is not included in the value returned by **clock**.

Under CMS, the base point for **clock** is the total processor time (**TOTCPU**) as returned by DIAGNOSE X'0C'. If the accumulated time is reset by the system operator after a call to **clock**, **clock** returns −1.0 thereafter because the amount of the processor time used can no longer be determined.

clock Measure Program Processor Time
(continued)

EXAMPLE

This example determines the processor time required to compute 1000 logarithms.

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

main()
{
   clock_t start, end;
   double index;

      /* time used before start of computation */
   start = clock();
   for(index = 1.0; index <= 1000.0; ++index)
      (void)log(index);
   end = clock();
   printf("Processor time used = %g seconds.\n",
          (end - start) / CLOCKS_PER_SEC);
}
```

RELATED FUNCTIONS

alarm, **time**

SEE ALSO

"Timing Functions" on page 2-11

close Close a File or Socket

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

int close(int fn);
```

DESCRIPTION

`close` closes a socket or a file opened for UNIX style I/O. `fn` is the file descriptor for the file or socket.

When `close` is called for an HFS file, any record locks for the file owned by the closing process are released. When the last open descriptor for a pipe or FIFO file is closed, any remaining data are lost. Also, when the last open descriptor for a file is closed, if its link count is now zero, the file is inaccessible and its disk space is released.

For a socket, `close` indicates that the program is finished using the socket; `close` breaks the connection between the socket descriptor and the socket. The status of undelivered output data is determined by the `SO_LINGER` socket option, as defined with the `setsockopt` function.

RETURN VALUE

`close` returns 0 if it is successful or −1 if it is unsuccessful. Even if `close` returns −1, any further attempts to use the file descriptor (unless it is reopened) will fail.

IMPLEMENTATION

Any unclosed files and sockets are automatically closed at normal program termination.

EXAMPLE

See the example for `open`.

RELATED FUNCTIONS

`fclose`, `fsync`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

_close Close an HFS File

DESCRIPTION

`_close` is a version of `close` designed to operate only on HFS files. `_close` runs faster and calls fewer other library routines than `close`. The `_close` function is used exactly like the standard `close` functions. Refer to `close` for a full description. The argument to `_close` must be the file descriptor for an open HFS file.

closedir Close Directory

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

int closedir(DIR *dir);
```

DESCRIPTION

closedir closes a directory. **dir** is the value returned by a previous call to **opendir** that opened the directory.

RETURN VALUE

closedir returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

See the example for **rewinddir**.

RELATED FUNCTIONS

opendir

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

clrerr Clear Error Flag and Return Status

SYNOPSIS

```
#include <stdio.h>

int clrerr(FILE *f);
```

DESCRIPTION

`clrerr` clears the error flag and end-of-file flag for the **`FILE`** object addressed by **`f`**.

RETURN VALUE

`clrerr` returns 0 if the error flag has been cleared or **`EOF`** if it could not be cleared. (The flag is not cleared if the operating system does not permit further use of the file or if the file is closed.)

EXAMPLE

```
#include <lcio.h>

   /* Clear the error flag for a file. */
int fixerr(FILE *f)
{
      /* Show error flag cleared.      */
   if (clrerr(f) != 0){
      printf("Error on %s is permanent. Program aborted.\n", fnm(f));
      abort();
   }
return 0;
}
```

RELATED FUNCTIONS

`clearerr`, **`ferror`**

SEE ALSO

"Error handling" on page 3-62

"I/O Functions" on page 2-13

cmsdfind Find the First CMS Fileid That Matches a Pattern

SYNOPSIS

```
#include <cmsio.h>

int cmsdfind(struct FILEINFO *info, char *pattern);
```

DESCRIPTION

cmsdfind searches for a CMS file whose name matches the pattern pointed to by **pattern**. The pattern is a string of the form *filename filetype filemode*, where one or more parts of the file ID can be an **''*''**. For example, **''* EXEC A''** is a pattern that matches all files with filetype EXEC on the A disk. Similarly, **''DATA * C''** is a pattern that matches all files with a filename of DATA on the C disk. If a match is found, **cmsdfind** fills in the **FILEINFO** structure.

You can specify **pattern** in any form of the **cms** or **xed** style that provides a file ID for a CMS disk file. For example, **''cms:*.exec.*''** is a pattern that matches all file IDS with a filetype of EXEC on any accessed disk.

The header file **<cmsio.h>** contains the structure **FILEINFO**. This structure is used for both **cmsdfind** and **cmsdnext**. The structure is defined as follows:

```
struct FILEINFO {
   char _[22];     /* reserved - do not alter   */
   char recfm;     /* file RECFM                */
   char _1;
   int lrecl;      /* file LRECL                */
   int norecs;     /* number of records in file */
   char name[21];  /* null-terminated fileid    */
};
```

RETURN VALUES

cmsdfind returns 0 if a match is found or a non-zero value if a match is not found.

If a match is found, the **name** element in the **FILEINFO** structure is filled in with the filename, filetype, and filemode of the matching fileid. The filename and filetype are padded on the right with blanks to eight characters. All parts of the fileid are in uppercase letters. The fileid is terminated with a null character.

The **recfm** field contains either 'F' or 'V', depending on the RECFM of the file. The **lrecl** field contains the LRECL of the file, and the **norecs** field contains the number of records in the file. Other file information can be retrieved with the **cmsstat** or **cmsstate** functions.

CAUTION

The first 22 characters in the **FILEINFO** structure are reserved for subsequent calls to **cmsdnext**. This area contains a copy of the pattern and a pointer to an internal control block. Do not alter this area.

Disk I/O is not permitted between calls to **cmsdfind** and **cmsdnext** if the I/O is to the same disk that is being searched by **cmsdfind** and **cmsdnext**. You also cannot use **cmspid** between **cmsdfind** and **cmsdnext** if these functions

cmsdfind Find the First CMS Fileid That Matches a Pattern
(continued)

refer to the same disk. Note that if you run VM/XA or VM/ESA, you can use **cmsffind** and **cmsfnext** in place of **cmsdfind** and **cmsdnext** to avoid these restrictions.

EXAMPLE

```
#include <cmsio.h>

int rc;
struct FILEINFO info;

   /* List all files with filename PROFILE. */
rc = cmsdfind(&info,"PROFILE * *");
while (rc == 0) {
   puts(info.name);
   rc = cmsdnext(&info);
}
```

RELATED FUNCTIONS

cmsdnext, **cmsffind**, **opendir**, **osdfind**

SEE ALSO

"File Management Functions" on page 2-16

cmsdnext Find the Next CMS Fileid Matching a Pattern

SYNOPSIS

```
#include <cmsio.h>

int cmsdnext(struct FILEINFO *info);
```

DESCRIPTION

`cmsdnext` finds the next CMS fileid matching a pattern previously defined in a call to **`cmsdfind`**. The **`info`** function is a **`FILEINFO`** structure that has been filled in by **`cmsdfind`**. If a match is found, then the information for the new match is placed in the structure pointed to by **`info`**. Refer to the description of **`cmsdfind`** for a listing of **`FILEINFO`**.

RETURN VALUES

Refer to the RETURN VALUE section of **`cmsdfind`**.

CAUTION

Refer to the CAUTION section of **`cmsdfind`**.

Disk I/O is not permitted between calls to **`cmsdfind`** and **`cmsdnext`** if the I/O is to the same disk that is being searched by **`cmsdfind`** and **`cmsdnext`**. You also cannot use **`cmspid`** between **`cmsdfind`** and **`cmsdnext`** if these functions refer to the same disk. Note that if you run VM/XA or VM/ESA, you can use **`cmsffind`** and **`cmsfnext`** in place of **`cmsdfind`** and **`cmsdnext`** to avoid these restrictions.

EXAMPLE

See the example for **`cmsdfind`**.

RELATED FUNCTIONS

`cmsdfind`

SEE ALSO

"File Management Functions" on page 2-16

cmsffind Find the First CMS Fileid That Matches a Pattern

SYNOPSIS

```
#include <cmsio.h>

int cmsffind(struct EXT_FILEINFO *info, char *pattern);
```

DESCRIPTION

cmsffind replaces **cmsdfind** for programs that will be used in VM/XA and VM/ESA operating systems.

cmsffind searches for the first CMS file whose name matches the pattern pointed to by **pattern**. The pattern is a string of the form *filename filetype filemode*, where one or more parts of the fileid can contain the special wildcard characters **''*''** (asterisk) and **''%''** (percent). The asterisk represents any number of characters, including 0. You can use as many asterisks as required in each part of the pattern. The percent character represents exactly one character.

If the pattern string omits any part of the fileid, an asterisk is used. For example, the pattern **''* EXEC A''** matches all files on the A disk with a filetype of EXEC. The pattern **''PROFILE *''** matches all files on any accessed disk having a filename of PROFILE. The pattern **''%%%% * *''** matches all files on all accessed disks whose filenames have exactly four characters. The pattern **''* HELP* Y2''** matches all files on the Y disk whose filemode number is 2 and which have a file type beginning with HELP.

If a match is found, **cmsffind** fills in the **EXT_FILEINFO** structure pointed to by **info**. This structure is defined in the **<cmsio.h>** header file. The structure is defined as

```
__alignmem struct EXT_FILEINFO {
   unsigned int internal_data[7];
   char Filename[9];
   char Filetype[9];
   char Filemode[3];
   char Format;
   unsigned int Lrecl;
   unsigned int Recs;
   unsigned int Blocks;
   char Date[9];
   char Time[9];
   char Fileid[21];
};
```

Subsequent CMS files whose fileids match the pattern may be found by calling **cmsfnext** using a pointer to the same **EXT_FILEINFO** structure. When all the fileids have been found, call **cmsfquit** to release the data reserved by **cmsffind**.

RETURN VALUES

cmsffind returns 0 if a match is found or a non-zero value if a match is not found.

cmsffind *(continued)* Find the First CMS Fileid That Matches a Pattern

After a successful match, **cmsffind** fills in the fields in the structure as follows:

Filename	contains the filename of the matching fileid.
Filetype	contains the filetype of the matching fileid.
Filemode	contains the filemode letter and number of the matching fileid.

The above three fields are NULL terminated and are not padded with blanks.

Format	is the record format of the file, either **F** or **V**.
Lrecl	is the maximum record length of the file.
Recs	is the number of records in the file.
Blocks	is the number of mini-disk blocks occupied by the file.
Date	is the date of the last update to the file in the form *MM/DD/YY*.
Time	is the time of the last update to the file in the form *HH/MM/SS*.
Fileid	is the fileid in the form *FILENAME FILETYPE FM* in a NULL-terminated string. The parts of *fileid* are separated by one blank space.

IMPLEMENTATION

cmsffind uses the DMSERP (Extract/Replace) routine of the CMS Callable Services Library. This routine is not available in VM/SP.

CAUTION

The **internal_data** field in the **EXT_FILEINFO** structure is reserved for subsequent calls to **cmsfnext**. Do not alter these data.

cmsffind invokes the DMSERP RESET function to initialize the Extract/Replace facility. If the program has previously used the Extract/Replace facility, any environmental information left over from that use will be destroyed. Thus, you cannot have more than one unterminated **cmsffind/cmsfnext** loop running simultaneously.

EXAMPLE

```
#include <stdio.h>
#include <cmsio.h>

int main(void)
{
   int match;

      /*  Type the names of all the files on the C disk whose   */
      /*   filenames have an "A" as the second character, whose */
      /*   filetype is "EXEC" followed by any characters, and   */
      /*   whose filemode number is 2.                          */
   match = cmsffind(&info, "%A* EXEC* C2");
```

cmsffind Find the First CMS Fileid That Matches a Pattern
(continued)

```
      while (match == 0) {
         puts(info.Fileid);
         match = cmsfnext(&info);
      }
      cmsfquit();
      return 0;
   }
```

RELATED FUNCTIONS

`cmsdfind`, **`cmsfnext`**, **`cmsfquit`**

SEE ALSO

"File Management Functions" on page 2-16

cmsfnext Find the Next CMS Fileid Matching a Pattern

SYNOPSIS

```
#include <cmsio.h>

int cmsfnext(struct EXT_FILEINFO *info);
```

DESCRIPTION

`cmsfnext` finds the next CMS fileid matching the pattern used in the previous call to `cmsffind`. The `info` function is a pointer to an `EXT_FILEINFO` structure used in a previous call to `cmsffind`. If a match is found, the information for the new match is stored in the `EXT_FILEINFO` structure, replacing whatever data were previously contained in the structure. Refer to `cmsffind` for a description of the data stored by `cmsfnext` in the `EXT_FILEINFO` structure.

RETURN VALUES

`cmsfnext` returns 0 if a matching fileid is found or a non-zero value if no matching fileid is found.

EXAMPLE

See the example for `cmsffind`.

RELATED FUNCTIONS

`cmsffind`, `cmsfquit`

SEE ALSO

"File Management Functions" on page 2-16

cmsfquit Release Data Held by cmsffind

SYNOPSIS

```
#include <cmsio.h>

void cmsfquit(void);
```

DESCRIPTION

cmsfquit releases the pattern-matching information that **cmsffind** created. Call **cmsfquit** after you have finished processing the CMS fileids returned by **cmsfnext**.

RETURN VALUE

cmsfquit has no return value.

CAUTION

cmsfquit invokes the DMSERP RESET function to reinitialize the Extract/Replace environment.

EXAMPLE

See the example for **cmsffind**.

RELATED FUNCTIONS

cmsffind, **cmsfnext**

SEE ALSO

"File Management Functions" on page 2-16

cmsstat Fill in a Structure with Information about a File

SYNOPSIS

```
#include <cmsstat.h>

int cmsstat(const char *path, struct cmsstat *buf);
```

DESCRIPTION

The **cmsstat** function fills in a **cmsstat** structure with system-dependent information about a file. For example, information is returned about the number of records in the file and the date the file was last modified. The file can be specified by any filename in the **cms**, **xed**, **ddn**, **sf**, or **sfd** style, except that VSAM files are not supported.

buf points to a **cmsstat** structure as defined in **<cmsstat.h>**. The **cmsstat** structure is defined as follows:

```
struct cmsstat {
   time_t st_mtime;        /* date last written              */
   unsigned st_type;       /* device type flags              */
   char st_flags;          /* access flags                   */
   char st_recfm;          /* RECFM                          */
   unsigned short st_lrecl; /* LRECL or terminal linesize     */
   int st_norecs;          /* number of logical records      */
                           /*  or number of terminal lines   */
   unsigned short st_bksiz; /* BLKSIZE                        */
   unsigned short st_vaddr; /* device virtual addr. in hex    */
   int st_dblks;           /* number of disk data blocks     */
   short st_dbksz;         /* minidisk blocksize             */
   char st_dlabl[7];       /* null-terminated disk label     */
   union  {
      struct  {            /* CMS fileid (if not S_OS)       */
         char name[9];     /* null-terminated CMS            */
                           /* filename or device name        */
         char type[9];     /* null-terminated CMS filetype   */
         char mode[3];     /* null-terminated CMS filemode   */
      } file;
      char dsn[45]; /* null-terminated MVS DSN (if S_OS)      */
   }  st_fid;       /* file data set name                     */
   char st_mem[9];  /* null-terminated member name (if S_ LIB) */
   char _ [5];      /* unused, a padding element              */
};
```

The **st_type** flag can have one of the following values or a combination of the following values (such as **S_OS** and **S_LIB**):

S_DUM	indicates a dummy file.
S_DISK	indicates a CMS disk file.
S_TERM	indicates a terminal.
S_TAPE	indicates a tape file.
S_UR	indicates a unit record device.

cmsstat Fill in a Structure with Information about a File
(continued)

`S_XED`	indicates a file is in XEDIT storage.
`S_OS`	indicates a file is on an MVS disk.
`S_3270`	indicates a 3270-type terminal.
`S_OSFORMAT`	indicates a file is in MVS format.
`S_LIB`	indicates a MACLIB, TXTLIB, or MVS PDS member.
`S_SFS`	indicates a Shared File System (SFS) file.
`S_SFSDIR`	indicates a (SFS) directory.

The **st_flags** access flag has the following values:

`S_RW`	indicates a file is read and write.
`S_WO`	indicates a file is write only.
`S_RWX`	indicates an extension of a read/write disk.
`S_RO`	indicates a file is read only.
`S_ROX`	indicates an extension of a read-only disk.
`S_EP`	indicates a file or directory is externally protected.
`S_NO`	indicates no authority on an SFS file or directory.

The following constant is defined for MVS disk files with spanned records:

```
S_LRECLX  0x8000
```

RETURN VALUES

If the file exists, **cmsstat** returns 0 and fills in the appropriate fields of the **cmsstat** structure. If the file does not exist or the filename is invalid, **cmsstat** returns −1.

CAUTION

You cannot use the **cmsstat** function to retrieve information about VSAM files.

Fileid or data set name information is always available after a successful call to **cmsstat**. However, the other fields in the **cmsstat** structure may not be useful for all types of files. For some files, some of the fields of the **cmsstat** structure are not meaningful. The values returned for each such field are as follows:

```
cmsstat.st_dlabl              ""
cmsstat.st_vaddr              0xffff
cmsstat.st_fid.file.name      ""
cmsstat.st_fid.file.type      ""
cmsstat.st_fid.file.mode      ""
cmsstat.st_mtime              (time_t) -1
cmsstat.st_lrecl              0xffff
cmsstat.st_bksiz              0xffff
cmsstat.st_norecs             -1
cmsstat.st_dblks              -1
```

cmsstat Fill in a Structure with Information about a File
(continued)

```
cmsstat.st_dbksz              -1
cmsstat.st_mem                ""
cmsstat.st_recfm              0xffff
```

If you specify the file with the style prefix **xed** and XEDIT is not active or the file is not found in XEDIT, the file is searched for on disk.

Fields in the **cmsstat** structure may have been modified, even if the function returns −1.

IMPLEMENTATION

For files on a CMS disk, the CMS FSSTATE macro is issued. For CMS (SFS) files and directories, the callable services library routine DMSEXIST is invoked.

EXAMPLE

```
#include <cmsstat.h>
#include <stdio.h>
#include <string.h>

#define UNDEF1 0xFFFF

main()
{
   struct cmsstat fileinfo;
   int rc;
   rc = cmsstat("cms:user maclib * (member memname)", &fileinfo);
   if (rc != 0)
      return rc;

      /* Check file type.                                       */
   if (fileinfo.st_type & S_LIB)
      puts("File is a member in a MACLIB. ");

      /* Print device's virtual address.                        */
   if (fileinfo.st_vaddr != UNDEF1)
      printf("Minidisk vaddr %x \n", fileinfo.st_vaddr);

      /* member name                                            */
   if (strlen(fileinfo.st_mem) == 0)
      puts("Member name not meaningful. ");
   else
      printf("Member name  %s \n", fileinfo.st_mem);
   return rc;
}
```

RELATED FUNCTIONS

stat, **osddinfo**, **osdsinfo**, **sfsstat**

cmsstat Fill in a Structure with Information about a File
(continued)

SEE ALSO

"File Management Functions" on page 2-16

cos Compute the Trigonometric Cosine

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double cos(double x);
```

DESCRIPTION

cos computes the trigonometric cosine of the value **x**. The **cos** function expects a value of **x** expressed in radians.

Because the cosine function is periodic, only the value of x mod 2π is used to compute the cosine. If **x** is very large, only a limited precision is left to represent x mod 2π. Thus, an error message is written for very large negative or positive arguments (see DIAGNOSTICS below).

RETURN VALUE

cos returns the value of the cosine of the argument **x**, provided that this value is defined and computable. The return value is of type **double**.

DIAGNOSTICS

For a very large argument (**x**> 6.7465e9), the function returns 0.0. In this case, the message "total loss of significance" is also written to **stderr** (the standard error file).

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

EXAMPLE

This example computes the cosine of a 60-degree angle using **cos**:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;

      /* The constant M_PI is declared in the  */
      /*  header file <lcmath.h>. 60 degrees   */
      /*  is pi/3 radians.                     */
   y = cos(M_PI/3);

   printf("cos(%f) =  %f\n",M_PI,y);
}
```

RELATED FUNCTIONS

sin, **tan**, **_matherr**

cos *(continued)* Compute the Trigonometric Cosine

SEE ALSO

“Mathematical Functions” on page 2-6

cosh Compute the Hyperbolic Cosine

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double cosh(double x);
```

DESCRIPTION

cosh computes the hyperbolic cosine of its argument **x**, as expressed by the following relation:

$$r = (e^{x} + e^{-x})/2$$

RETURN VALUE

cosh returns the value of the hyperbolic cosine of the argument **x**, provided that this value is defined and computable. The return value is a double-precision, floating-point number.

DIAGNOSTICS

For **x** with an absolute value too large to be represented, the function returns **HUGE_VAL**. The run-time library writes an error message to **stderr** (the standard error file).

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

EXAMPLE

This example computes the hyperbolic secant of a number using **cosh**:

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

#define VALUE 3.76182

main()
{
   double secant_h;

      /* The hyperbolic secant of VALUE is 1 divided  */
      /* by the hyperbolic cosine of VALUE.           */
   secant_h = 1 / cosh(VALUE);

   printf("1 / cosh(%f) = %f\n", VALUE, secant_h);
}
```

RELATED FUNCTIONS

sinh, **tanh**, **_matherr**

cosh Compute the Hyperbolic Cosine
(continued)

SEE ALSO

"Mathematical Functions" on page 2-6

creat Create and Open a File for UNIX Style I/O

SYNOPSIS

```
#include <sys/types.h>
#include <fcntl.h>

int creat(const char *name, mode_t mode);
```

DESCRIPTION

creat creates a file and opens it in write-only mode, if the file does not already exist. **creat** is a special case of the **open** function. **creat(name, mode)** is equivalent to **open(name, O_WRONLY | O_CREAT | O_TRUNC, mode)**. The **mode** function defines file permissions for an HFS file and is otherwise ignored. You do not need to specify **mode** unless you are opening an HFS file.

RETURN VALUE

creat returns the file number if the file is successfully created and opened or −1 if it is not.

CAUTION

If you use **creat** to create a file and then close it without writing any characters, the file may not exist after it is closed. Refer to "IBM 370 I/O Concepts" on page 3-7 for more discussion of this point.

EXAMPLE

```
#include <sys/typed.h>
#include <fcntl.h>

int cardfile;

   /* Create and open the file.       */
cardfile = creat("cards",S_IRWXU);
   .
   .     /* file processing statements */
   .
close(cardfile);    /* Close the file. */
```

RELATED FUNCTIONS

mkdir, **open**, **tmpfile**, **umask**

SEE ALSO

"Opening Files" on page 3-27

Chapter 19, "Introduction to POSIX" in *SAS/C Library Reference, Volume 2*

"I/O Functions" on page 2-13

ctermid Get Filename for the Terminal

SYNOPSIS

```
#include <lcio.h>

char *ctermid(char *termid);
```

The synopsis for the POSIX implementation is as follows:

```
#include <unistd.h>

char *ctermid(char *termid);
```

This set of header files requires the definition of an appropriate feature test macro. See "Feature Test Macros" on page 1-6 for more information.

DESCRIPTION

ctermid returns the filename of the user's terminal. **string** is the terminal filename.

The argument to **ctermid** should be **NULL**, or it should address a character array (**termid**) whose size is at least **L_ctermid**. The symbol **L_ctermid** is defined in the header file **<lcio.h>** (or **<stdio.h>** if an appropriate feature test macro is defined).

If the argument is not **NULL**, the filename (followed by '\0') is copied into the area addressed by **termid**. If the argument is **NULL**, the filename can only be accessed by using the value returned by **ctermid**.

RETURN VALUE

The return value is a pointer to the filename. If the argument to **ctermid** is **NULL** (the norm), the return value is in static storage and may be overlaid by the next call to **ctermid**.

CAUTION

If a noninteractive program calls **ctermid**, an attempt to open the filename returned by **ctermid** may fail.

EXAMPLE

This example sends a message to a terminal:

```
#include <lcio.h>

main()
{
      /* Open the terminal for writing.                  */
   FILE *termfile;

      /* Assign name of interactive terminal to termfile. */
   termfile = fopen(ctermid(NULL), "w");
```

ctermid Get Filename for the Terminal
(continued)

```
      if (!termfile) {
         printf("File could not be opened.n", stderr);
         exit(1);
      }

         /* Print message to interactive terminal.           */
      fprintf(termfile, "This is a test message.n");
      fclose(termfile);
   }
```

RELATED FUNCTIONS

`ttyname`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

ctime Convert Local Time Value to Character String

POSIX ISO ANSI UNIX

SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *timep);
```

DESCRIPTION

ctime converts a **time_t** value (as returned by the **time** function) to a printable character string and returns the address of the first character of the string. The string has the form **wkd mon dd hh:mm:ss yyyy\n**, for example, **Thu Oct 10 16:49:07 1985\n**. The length of the string is always 25. (The day of the month is padded on the left with blanks to two characters if necessary; the hours, minutes, and seconds are padded with 0s.)

ctime is affected by time zone information contained in the TZ environment variable.

RETURN VALUE

ctime returns a pointer to the formatted local date and time.

CAUTION

The pointer returned by **ctime** may reference **static** storage, which may be overwritten by the next call to **asctime** or **ctime**.

IMPLEMENTATION

ctime(timep) is implemented as **asctime(localtime(timep))**.

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main ()
{
   time_t timeval;
   time(&timeval);
   printf("The current time and date are: %s",
           ctime(&timeval));
}
```

RELATED FUNCTIONS

asctime

SEE ALSO

"Timing Functions" on page 2-11

cuserid Get Current Userid

SYNOPSIS

```
#include <lcio.h>

char *cuserid(char *name);
```

DESCRIPTION

cuserid gets the userid associated with the current job or interactive session.

The argument to **cuserid** should be **NULL**, or it should address a character array (**name**) whose size is at least **L_cuserid**. The symbol **L_cuserid** is defined in the header file **<lcio.h>**.

If the argument is not **NULL**, the userid (followed by '\0') is copied into **name**. If the argument is **NULL**, the userid can only be accessed by using the value returned by **cuserid**.

Under CMS and TSO, the userid is defined by VM or TSO, respectively. Under MVS batch, the userid is defined only if RACF (or a similar product) is installed.

RETURN VALUE

cuserid returns a pointer to the userid. If the argument to **cuserid** is **NULL**, the return value is in static storage and may be overlaid by the next call to **cuserid**.

DIAGNOSTICS

If no userid can be determined, **cuserid** returns a pointer to a string with length 0.

IMPLEMENTATION

The size of the string where the userid is stored is determined by the constant **L_cuserid**, defined in the header file **<lcio.h>**.

Under CMS, the userid is returned by the VM control program (CP).

EXAMPLE

```
#include <lcio.h>
#include <time.h>
#include <stdlib.h>

main()
{
   FILE *logfile;
   char username[L_cuserid];
   time_t now;

      /* Open SYSLOG to add data to the end of the file. */
   logfile = fopen("ddn:SYSLOG", "a");
   if (!logfile){
      puts("Failed to open log file.");
      exit(EXIT_FAILURE);
   }
```

cuserid Get Current Userid
(continued)

```
      cuserid(username);                     /* Get userid. */
      time(&now);
      fprintf(logfile, "File logfile last accessed by %s on %s",
              username, ctime(&now));
      fclose(logfile);
   }
```

SEE ALSO

"System Interface and Environment Variables" on page 2-17

difftime Compute the Difference of Two Times

SYNOPSIS

```
#include <time.h>

double difftime(time_t time2, time_t time1);
```

DESCRIPTION

`difftime` computes the difference **`time2 - time1`** in seconds, where **`time2`** and **`time1`** are values of type **`time_t`** (such as those returned by the **`time`** function).

RETURN VALUE

`difftime` returns the difference between the two times in seconds.

CAUTION

`difftime` is implemented as a macro. If its arguments are not of type **`time_t`**, the results are not meaningful.

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main()
{
   double x;
   time_t last_written;          /* time data last written */
                                 /* time(&last_written);   */

      /* if data are more than 1 week old                  */
   if ((x = difftime(time(NULL), last_written)) > 7 * 86400)
      puts("Warning: These data are obsolete");
   else printf("Data was last accessed %lf seconds ago.\n", x);
}
```

RELATED FUNCTIONS

`time`

SEE ALSO

“Timing Functions” on page 2-11

div Integer Division

SYNOPSIS

```
#include <stdlib.h>

div_t div(int numer, int denom);
```

DESCRIPTION

div computes the quotient and remainder of **numer** divided by **denom**.

RETURN VALUE

div returns a structure of type **div_t**, which contains both the quotient and remainder. The definition of the **div_t** type is

```
typedef struct {
int rem;
int quot;
} div_t;
```

The return value is such that

```
numer == quot * denom + rem
```

The sign of **rem** is the same as the sign of **numer**.

EXAMPLE

This example converts an angle in radians to degrees, minutes, and seconds:

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

main()
{
   double rad, angle;
   int deg, min, sec;
   div_t d;

   puts(" Enter any angle in radians: ");
   scanf("%lf", &rad);

      /* Convert angles to seconds and discard fraction. */
   angle = rad * (180 * 60 * 60)/M_PI;

   sec = angle;
   d = div(sec, 60);
   sec = d.rem;
```

div Integer Division
(continued)

```
         d = div(d.quot, 60);
         min = d.rem;
         deg = d.quot;

         printf("%f radians = %d degrees, %d', %d", n", rad, deg,
                min, sec);
      }
```

RELATED FUNCTIONS

`ldiv`

SEE ALSO

"Mathematical Functions" on page 2-6

dup Duplicate File Descriptor

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

int dup(int filedes);
```

DESCRIPTION

dup duplicates an OpenEdition file descriptor to the lowest numbered available file descriptor. **filedes** is the original file descriptor. The new descriptor has the same file position as the original file descriptor and shares any record locks.

RETURN VALUE

dup returns a file descriptor if successful and it returns a −1 if it is not successful.

EXAMPLE

This example invokes the shell command **tr** to translate a file to lowercase and copy it to **stdout**. The filename is specified on the command line. The **dup** function assigns the file to standard input before using **execlp** to invoke the **tr** command. This example should be compiled with the **posix** option and run under the OpenEdition shell:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

main(int argc, char *argv[]) {
   int input;
   int newfd;
      /* If no argument, input is stdin.            */
   if (argc > 1) {
      input = open(argv[1], O_RDONLY);
      if (input < 0) {
         perror("open error");
         exit(EXIT_FAILURE);
      }
         /* If input is already fd 0, no dup needed. */
      if (input != STDIN_FILENO) {
            /* Close standard input. */
         close(STDIN_FILENO);
            /* Duplicate to lowest avail fd (0).     */
         newfd = dup(input);
         if (newfd != 0) {    /* should not occur    */
            fputs("Unexpected dup error.n", stderr);
            exit(EXIT_FAILURE);
         }
```

dup Duplicate File Descriptor
(continued)

```
                /* Close original fd. */
              close(input);
           }
        }
        execlp("tr", "tr", "[:upper:]", "[:lower:]");
        perror("exec error");  /* exec must have failed.  */
        exit(EXIT_FAILURE);
     }
```

RELATED FUNCTIONS

`dup2`, **`fcntl`**

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

“I/O Functions” on page 2-13

dup2 Specify Duplicate File Descriptor

SYNOPSIS

```
#include <unistd.h>

int dup2(int fd1, int fd2);
```

DESCRIPTION

dup2 returns a file descriptor with the value of **fd2**. The **fd2** function refers to the same file as **fd1**. If **fd2** is an open file descriptor and does not equal **fd1**, **fd2** is closed before it is duplicated. **fd1** must be the file number of an open HFS file.

RETURN VALUE

dup2 returns a file descriptor if successful and it returns a −1 if it is not successful.

EXAMPLE

This example invokes the shell command **tr** to translate a file to lowercase and copy it to **stdout**. The filename is specified on the command line. The **dup2** function assigns the file to standard input before using **execlp** to invoke the **tr** command. This example has the same purpose as the **dup** example but is less complicated due to the use of **dup2**. This example should be compiled with the **posix** option and run under the OpenEdition shell:

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

main(int argc, char *argv[]) {
   int input;
   int newfd;
      /* If no argument, input is stdin.            */
   if (argc > 1) {
      input = open(argv[1], O_RDONLY);
      if (input < 0) {
         perror("open error");
         exit(EXIT_FAILURE);
      }
         /* sup input to standard input             */
      newfd = dup2(input, STDIN_FILENO);
      if (newfd < 0) {
         perror("dup2 error");
         exit(EXIT_FAILURE);
      }
         /* Close original fd.                      */
      if (newfd != input) {
         close(input);
      }
   }
```

dup2 Specify Duplicate File Descriptor
(continued)

```
      execlp("tr", "tr", "[:upper:]", "[:lower:]");
      perror("exec error");  /* exec must have failed. */
      exit(EXIT_FAILURE);
   }
```

RELATED FUNCTIONS

dup, **fcntl**

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"I/O Functions" on page 2-13

ecbpause Wait for Signal

SYNOPSIS

```
#include <lcsignal.h>

int ecbpause(int mask, size_t listsize, struct _ecblist *ecblist);
```

DESCRIPTION

ecbpause delays program execution until it receives a C signal or an Event Control Block (ECB) is posted. **mask** specifies the mask of signals managed by SAS/C to be blocked while execution is delayed. **listsize** specifies the number of **_ecblist** structures addressed by the **ecblist** argument.

ecblist is the address of an array of structures, each of which represents one or more contiguous ECBs. 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.

The declaration for the **_ecblist** structure is:

```
struct _ecblist {
   size_t count;
   unsigned *ecbarr;
}
```

The ECB list for **ecbpause** is passed by the **_ecblist** structure for several reasons. It enables a static ECB list to be used in many cases since individual ECBs can easily be removed by setting their **count** member to 0. For applications that have a large number of ECBs, the **_ecblist** structure facilitates organizing them into arrays; this method may slightly improve the performance of **ecbpause** because fewer memory accesses are required to determine the addresses of all the ECBs.

ecbpause returns to its caller when one of the following events occurs:

- an unblocked signal occurs. A handler for the signal is called after the signal mask is restored but before control returns to the caller from **ecbpause**.
- an ECB is POSTed.

Several conditions for completion of the **ecbpause** function may occur simultaneously or nearly simultaneously. In such cases, a signal handler may be called even though an ECB was POSTed before or during arrival of the signal. On return from **ecbpause**, any number of ECBs may be POSTed, and more than one signal handler may have been called if the signal mask permits it.

RETURN VALUE

ecbpause returns the **errno** value **EINTR** if an unblocked signal was pending at the completion of its wait. Otherwise, it returns zero.

CAUTIONS

ecbpause does not clear any ECBs addressed with the **ecblist** argument. It is the caller's responsibility to clear the ECBs after a POST and to initialize them to zero or to some other suitable value.

The value returned by **ecbpause** may not be completely reliable. An ECB may have been POSTed even though a signal was detected, and a signal may have been received after **ecbpause** was awakened by a POST but before return

ecbpause Wait for Signal

(continued)

to the user program was completely effected.

ecbpause does not permit the caller to change the signal mask for any signals managed by OpenEdition. Programs that handle OpenEdition signals should use the **ecbsuspend** function instead.

IMPLEMENTATION

ecbpause builds a standard OS ECB list for the ECBs indicated by its arguments, in addition to an ECB used internally by signal handling, and issues the OS WAIT macro to wait for a single ECB to be POSTed.

EXAMPLE

Wait for an alarm signal or for a POST representing a reply from the MVS operator. This example assumes that **SIGALRM** is not managed by OpenEdition. See the **sigsuspend** example for a version that works regardless of whether **SIGALRM** is managed by OpenEdition.

```
#include <lcsignal.h>
#include <ctype.h>

 /* flag for SIGALRM handler   *
static int toolate = 0;
static void timesup(int signum);

int confirm()
{
   unsigned ECB = 0;
   struct _ecblist myECBs = {1, 0 } ;
   char reply;

   /* Set up ECB list for single ECB. */
   myECBs.ecbarr = &ECB;

   /* Issue WTOR macro via assembler. */
        /* subroutine (not shown)     */
   wtor("Reply U to confirm request.", &reply, &ECB);

   /* Catch SIGALRM signal.      */
   signal(SIGALRM, &timesup);
   toolate = 0;
   /* Wait 2 minutes for reply.  */
   alarm(120);
   ecbpause(0, 1, &myECBs);
   /* Cancel alarm.              */
   alarm(0);
```

ecbpause Wait for Signal
(continued)

```
   /* Restore default alarm handling. */
   signal(SIGALRM, SIG_DFL);
      /* If the ECB was posted,     */
      /*  return whether OK given.  */
      if (ECB & 0x40000000)
         return toupper(reply) == 'U';
      /* If we ran out of time,     */
      /* call asm to delete reply.  */
      else if(toolate){
         dom();
        /* Tell caller not to do it.  */
        puts("No reply received, treated as permission denied. ");
        return 0;
      }
}

static void timesup(int signum) {
   toolate = 1;
   return;
}
```

RELATED FUNCTIONS

ecbsuspend, **sigpause**, **sigsuspend**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

ecbsuspend Suspend Execution Until a Signal or Post

SYNOPSIS

```
#include <lcsignal.h>

int ecbsuspend(sigset_t *mask, size_t listsize,
struct _ecblist *ecblist);
```

DESCRIPTION

ecbsuspend delays program execution until it receives a C signal or until an Event Control Block (ECB) is posted. **mask** specifies the set of signals to be blocked while execution is delayed. **listsize** specifies the number of **_ecblist** structures addressed by the **ecblist** argument.

ecblist is the address of an array of structures, each of which represents one or more contiguous ECBs. 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.

The declaration for the **_ecblist** structure is

```
struct _ecblist {
   size_t count;
   unsigned *ecbarr;
}
```

The ECB list for **ecbsuspend** is passed by the **_ecblist** structure for several reasons. It enables a static ECB list to be used in many cases because individual ECBs can easily be removed by setting their **count** member to 0. For applications that have a large number of ECBs, the **_ecblist** structure facilitates organizing them into arrays; this method may slightly improve the performance of **ecbsuspend** because fewer memory accesses are required to determine the addresses of all the ECBs.

ecbsuspend returns to its caller when one of the following events occurs:

- an unblocked signal occurs. A handler for the signal is called after the signal mask is restored but before control returns to the caller from **ecbsuspend**.
- an ECB is POSTed.

Several conditions for completion of the **ecbsuspend** function may occur simultaneously or nearly simultaneously. In such cases, a signal handler may be called even though an ECB was POSTed before or during arrival of the signal. On return from **ecbsuspend**, any number of ECBs may be POSTed, and more than one signal handler may have been called if the signal mask permits it.

RETURN VALUE

ecbsuspend returns −1 if it was terminated due to receipt of a signal or due to an error. (**errno** is set to **EINTR** if the cause was a signal.) **ecbsuspend** returns 0 if it returned because an ECB was posted.

CAUTIONS

ecbsuspend does not clear any ECBs addressed by the **ecblist** argument. It is the caller's responsibility to clear the ECBs after a POST and to initialize them to zero or to some other suitable value.

The value returned by **ecbsuspend** may not be completely reliable. An ECB

ecbsuspend Suspend Execution Until a Signal or Post
(continued)

may have been POSTed even though a signal was detected, and a signal may have been received after **ecbsuspend** was awakened by a POST but before return to the user program was completely effected.

IMPLEMENTATION

ecbsuspend builds a standard OS ECB list for the ECBs indicated by its arguments, in addition to an ECB used internally by signal handling, and issues the OS WAIT macro to wait for a single ECB to be POSTed. If OpenEdition is active, the OpenEdition **mvspause** system call is used in place of an OS WAIT.

EXAMPLE

Wait for an alarm signal or for a POST representing a reply from the MVS operator:

```
#include <lcsignal.h>
#include <ctype.h>

/* flag for SIGALRM handle */
static int toolate = 0;
static void timesup(int signum);

int confirm()
{
   unsigned ECB = 0;
   struct _ecblist myECBs = {1, 0 } ;
   char reply;
   sigset_t nosigs;

    /* Set up ECB list for single ECB.  */
   myECBs.ecbarr = &ECB;

   /* Issue WTOR macro via assembler.  */
   /* subroutine (not shown)           */
   wtor("Reply U to confirm request.", &reply, &ECB);

   /*Catch SIGALRM signal.             */
   signal(SIGALRM, &timesup);
   /* Wait 2 minutes for reply.        */
   toolate = 0;
   alarm(120);
    /* Set no sigs blocked for suspend. */
   sigemptyset(&nosigs);
   ecbsuspend(&nosigs, 1, &myECBs);
   /* Cancel alarm.                    */
   alarm(0);
```

ecbsuspend Suspend Execution Until a Signal or Post
(continued)

```
      /* Restore default alarm handling.  */
      signal(SIGALRM, SIG_DFL);
         /* If the ECB was posted,           */
         /* return whether OK given.         */
         if (ECB & 0x40000000)
           return toupper(reply) == 'U';
         /* If we ran out of time,           */
         /* call asm to delete reply.       */
         else if(toolate){
            dom();
            /* Tell caller not to do it.       */
            puts("No reply received, treated as permission denied. ");
            return 0;
         }
   }

   static void timesup(int signum) {
      toolate = 1;
      return;
   }
```

RELATED FUNCTIONS

ecbpause, **sigpause**, **sigsuspend**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

erf Compute the Error Function

SYNOPSIS

```
#include <lcmath.h>

double erf(double x);
```

DESCRIPTION

erf computes the error function of its argument **x**. The value returned by **erf** is defined by this equation:

$$\mathrm{erf}(x) = (2 / \sqrt{\pi})\int_0^x e^{-z^2}\,dz$$

RETURN VALUE

erf returns the error function of its argument.

EXAMPLE

This example computes the error function using **erf**:

```
#include <stdio.h>
#include <lcmath.h>

#define SVECTOR .7854

main()
{
   double erfv;
   erfv = erf(SVECTOR);
   printf("erf(%f) = %f\n", SVECTOR, erfv);
}
```

RELATED FUNCTIONS

erfc, **_matherr**

SEE ALSO

"Mathematical Functions" on page 2-6

erfc Compute the Complementary Error Function

UNIX

SYNOPSIS

```
#include <lcmath.h>

double erfc(double x);
```

DESCRIPTION

erfc computes the complementary error function of its argument **x**. The value returned by **erfc** is defined by this equation:

$$\mathrm{erfc}(x) = (2\ /\ \sqrt{\pi})\int_{x}^{\infty} e^{-z^2}\,dz$$

Note the following:

$$\mathrm{erf}(x) + \mathrm{erfc}(x) = 1$$

RETURN VALUE

erfc returns the complementary error function of its argument.

DIAGNOSTICS

An error message is written to the standard error file (**stderr**) by the run-time library if **x** exceeds the value 13.30619656013802. In this case, the function returns 0.0.

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

EXAMPLE

This example computes the complementary error function using **erfc**:

```
#include <stdio.h>
#include <lcmath.h>

#define SVECTOR .7854

main()
{
   double erfcv;
   erfcv = erfc(SVECTOR);
   printf("erfc(%f) = %f\n", SVECTOR, erfcv);
}
```

RELATED FUNCTIONS

erf, **_matherr**

SEE ALSO

"Mathematical Functions" on page 2-6

exit Terminate Execution

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

void exit(int code);
```

DESCRIPTION

exit terminates the program after closing all files. The integer argument **code** is used in an implementation-defined way to indicate the results of execution. Usually, a code of **EXIT_SUCCESS** indicates successful execution, and a code of **EXIT_FAILURE** indicates an unsuccessful execution.

RETURN VALUE

Control does not return from **exit**.

ERRORS

User ABEND 1203 or 1209 may be issued by **exit** if memory management data areas have been overlaid.

PORTABILITY

exit is generally portable, although any meaning associated with the **code** argument is not. Some systems only treat the last 8 bits of the **code** as significant, so codes from 0 to 255 are recommended for maximum portability.

Many C implementations also support a routine named **_exit** to terminate execution without closing files. This routine is available only when OpenEdition under MVS is installed.

IMPLEMENTATION

exit is implemented as a **longjmp** to a target defined in the library routine that calls **main**. Therefore, it can be intercepted with **blkjmp**.

On IBM's 370 system, **EXIT_SUCCESS** is 0 and **EXIT_FAILURE** is 16. The exit code is used as an MVS or CMS return code. Under MVS, a code that is not between 0 and 4095 is changed to 4095.

exit Terminate Execution
(continued)

EXAMPLE

This example shows how to exit a program if it is not called with a valid input filename:

```
#include <stdlib.h>
#include <stdio.h>

main(int argc, char *argv[])
{
   FILE *f;

   if (argc > 1){
      f = fopen(argv[1],"r");
      if (f == NULL){
         fprintf(stderr,
                 "Can't open file \"%s\"\n", argv[1]);
         exit(EXIT_FAILURE);
      }
      fclose(f);
      puts("File successfully opened and closed.");
      exit(EXIT_SUCCESS);
   }
   else{
      fprintf(stderr,"No file specified.\n");
      exit(EXIT_FAILURE);
   }
}
```

RELATED FUNCTIONS

abort, **atexit**, **blkjmp**, **coexit**, **_exit**

SEE ALSO

Appendix 5, “Using the indep Option for Interlanguage Communication,” in *SAS/C Compiler and Library User’s Guide*

“Program Control Functions” on page 2-9

exp Compute the Exponential Function

SYNOPSIS

```
#include <math.h>

double exp(double x);
```

DESCRIPTION

exp computes the exponential function of its argument **x**. The result is **e** to the **x** power, where **e** is the base of natural logarithms, 2.71828 . . ..

The exponential function is the inverse of the natural logarithm and is expressed by this relation:

$r = e^x$

x is a double-precision, floating-point number.

RETURN VALUE

exp returns the exponential function of its argument **x**, expressed as a double-precision, floating-point number.

DIAGNOSTICS

If **x** is too large and the ensuing result is so large that it cannot be represented, **exp** returns **HUGE_VAL**. In this case, the run-time library writes an error message to the standard error file (**stderr**).

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

EXAMPLE

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

#define XVAL 10.0

main()
{
   double y;

      /* Compute exponent function. */
   y = exp(-XVAL);
   printf("exp(-%f) = %f\n", XVAL, y);
}
```

RELATED FUNCTIONS

_matherr

exp Compute the Exponential Function
(continued)

SEE ALSO

"Mathematical Functions" on page 2-6

fabs Floating-Point Conversion: Absolute Value

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double fabs(double y);
```

DESCRIPTION

fabs takes the absolute value of a double-precision, floating-point number.

RETURN VALUE

fabs returns the absolute value of the argument **y**. Both the operand **y** and the result are of type **double**.

IMPLEMENTATION

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

EXAMPLE

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

main()
{
  double a, b, c;

  puts("Enter values for a & b");
  scanf("%lf %lf", &a, &b);
  c = fabs(a-b);
  printf("The absolute value of their difference = %f", c );
}
```

RELATED FUNCTIONS

abs, **labs**

SEE ALSO

"Mathematical Functions" on page 2-6

fattr Return File Attribute Information

SYNOPSIS

```
#include <lcio.h>

const struct fattrib *fattr(FILE *f);
```

DESCRIPTION

The **fattr** function returns a pointer to a structure containing information about the attributes of the stream associated with the **FILE** object addressed by the **f** argument. The return value is a pointer to a **fattrib** structure as defined in **<lcio.h>**. Note that the returned pointer is a **const** pointer, so you can inspect the structure fields but not modify them.

The **fattrib** structure is defined as

```
struct fattrib {
   unsigned long props;        /* file properties           */
   unsigned short abilities;   /* how file can be used      */
   unsigned short reclen;      /* maximum record length     */
                               /* (0 if no limit)           */
   unsigned short blksize;     /* block size or VSAM        */
                               /* control interval size     */
                               /* (0 if not meaningful)     */
   unsigned short keylen;      /* key length or 0           */
   unsigned short keyoff;      /* record offset of key or 0 */
   char am[5];                 /* name of access method     */
   char _1[1];                 /* reserved                  */
   long _2[10];                /* reserved                  */
} ;
```

The **props** field of the **fattrib** structure is a bit string composed of bits specifying whether the file has a number of specific properties. The **abilities** field is a bit string specifying whether the file supports specific file operations. Note that not all bits of these fields are defined and that undefined bits do not have to be set to 0.

The bit definitions for the **props** field are

_Fappend	indicates that the stream was opened for append.
_Fcommit	indicates that **commit=yes** was specified or defaulted.
_Fdupkey	indicates that the file permits records with duplicate keys.
_Ffixed	indicates that all records have the same size.
_Fgeneric	indicates that the file supports generic key searches.
_Finexact	indicates that the file supports inexact searches.
_Fkeyed	indicates a stream opened for keyed access.
_Fphyskey	indicates that file keys are physically recorded. This is set for a KSDS but not an ESDS or RRDS.
_Fprintform	indicates that the file has print format (has RECFM A or is a CMS LISTING file).
_Fprint	indicates that **print=yes** is in effect for the file.

fattr Return File Attribute Information
(continued)

_Fprocess	indicates that a hierarchical file system (HFS) file associated with the current process is accessible in any child process created by **fork** or **exec**, unless **fcntl** has prevented access.
_Frelative	indicates that file positions are expressed as relative-byte offsets.
_Fshared	indicates that an HFS file can be shared with another process. Some library optimizations of seek operations are disabled.
_Fspanbuf	indicates that records may be longer than the buffer size.
_Fspanrec	indicates that a single record can be written in more than one physical block.
_Fstream	indicates that record boundaries are ignored.
_Fterm	indicates a terminal file.
_Ftext	indicates a text stream.
_Ftrunc	indicates that **trunc=yes** was specified or defaulted.

The bit definitions for the **abilities** field are

_Fcan_delete	indicates that the file supports deletion of records with **kdelete**.
_Fcan_grow	indicates that new records can be added to the file.
_Fcan_read	indicates that the stream can be read.
_Fcan_rewind	indicates that the stream supports seeking to the start of file.
_Fcan_setbuf	Indicates that an HFS file or socket supports **setbuf** or **setvbuf** to define an I/O buffer. **setbuf** and **setvbuf** can only be the first file operation. **_Fcan_setbuf** remains in effect while a file is open.
_Fcan_search	indicates that the file supports key searches with **ksearch**.
_Fcan_seek	indicates that the stream supports positioning with **fsetpos**.
_Fcan_skend	indicates that the stream supports seeking to the end of file.
_Fcan_tell	indicates that the file supports position inquiries by **fgetpos** or **kgetpos**.
_Fcan_write	indicates that the stream can be written.

The **reclen** field of the **fattrib** structure includes space for the key in a keyed file. Thus, for an ESDS or RRDS opened for keyed access, the returned **reclen** value is 4 bytes greater than the maximum physical record size. In all cases in which a non-zero value is returned for **reclen=**, the value returned is the same as would be specified by the **reclen=** amparm; that is, the value is the same as the amount of storage required to read the largest possible record in the file.

fattr Return File Attribute Information

(continued)

RETURN VALUES

The **fattr** function returns a pointer to an **attrib** structure for the file. If the argument to **fattr** addresses a closed file object, a pointer to a dummy **fattrib** structure is returned with the **abilities** field equal to 0. If the argument to **fattr** is an invalid **FILE** pointer, the results are unpredictable.

EXAMPLE

```
#include <lcio.h>
#include <stdio.h>

main()
{
  FILE *outfile;

  outfile = afopen("tso:userid.test", "w", "",
                   "recfm=v, reclen=132");

  if (fattr(outfile)->props & _Fprintform)
     putc('\f', outfile);

  fclose(outfile);
}
```

RELATED FUNCTIONS

cmsstat, **fstat**, **osddinfo**, **osdsinfo**, **stat**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fchmod Change Directory or File Mode

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int fchmod(int fileDescriptor, mode_t mode);
```

DESCRIPTION

fchmod changes the file permission flags for the directory or file specified by **fileDescriptor**. The **mode** argument can be any combination of the following symbols, which are defined in **<stat.h>**:

S_ISUID	sets the user ID for execution. When the specified file is processed through an **exec** function, the user ID of the process is also set for execution.
S_ISGID	sets group ID for execution. When the specified file is processed through an **exec** function, the group ID of the process is also set for execution.
S_ISVTX	specifies shared text.
S_IRUSR	sets file owner permission to read.
S_IWUSR	sets file owner permission to write.
S_IXUSR	sets file owner permission to execute.
S_IRWXU	sets file owner permission to read, write, and execute.
S_IRGRP	sets group permission to read.
S_IWGRP	sets group permission to write.
S_IXGRP	sets group permission to execute.
S_IRWXG	sets group permission to read, write, and execute.
S_IROTH	sets general permission to read.
S_IWOTH	sets general permission to write.
S_IXOTH	sets general permission to execute.
S_IRWXO	sets general permission to read, write, and execute.

RETURN VALUE

fchmod returns 0 if it is successful. If unsuccessful, a −1 is returned.

EXAMPLE

This example changes a file whose file number is passed so that it can be executed by any user who can read it:

```
#include <sys/types.h>
#include <sys/stat.h>

int fchexec(int fd) {
   struct stat stat_data;
   mode_t newmode;
```

fchmod Change Directory or File Mode
(continued)

```
      int rc;

      rc = fstat(fd, &stat_data);
      if (rc != 0) {
         perror("fstat failure");
         return -1;
      }
      newmode = stat_data.st_mode;
      if (newmode & S_IRUSR) newmode |= S_IXUSR;
      if (newmode & S_IRGRP) newmode |= S_IXGRP;
      if (newmode & S_IROTH) newmode |= S_IXOTH;

         /* If the mode bits changed, make them effective. */
      if (newmode != stat_data.st_mode) {
         rc = fchmod(fd, newmode);
         if (rc != 0) perror("fchmod failure");
         return rc;
      }
      return(0);    /* No change was necessary.            */
   }
```

RELATED FUNCTIONS

`chmod`, **`chown`**

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

fclose Close a File

SYNOPSIS

```
#include <stdio.h>

int fclose(FILE *f);
```

DESCRIPTION

fclose disassociates the **FILE** object addressed by **f** from the associated external file after writing out any buffered output data. (By definition, standard I/O is always buffered.)

RETURN VALUE

fclose returns 0 if the file is closed successfully. It returns a nonzero value if it is not closed successfully.

If **fclose** fails, you cannot use the **FILE** object addressed by **f**. The file is closed to you, and you have to reopen it. Because **fclose** flushes the output buffer, an error can occur in buffer flushing with the result that **fclose** has effectively failed. Even in this case, further use of the **FILE** pointer is not possible.

DIAGNOSTICS

Any attempt to use a **FILE** pointer (except as an argument to **freopen**) after the file is closed is erroneous.

IMPLEMENTATION

All open **FILE**s are automatically closed at normal program termination.

USAGE NOTES

Because most implementations limit the number of files that can be open at one time, **fclose** is useful in programs that deal with multiple files. Files that are unused can be closed to save memory space and to keep within any constraints on the number of files that may be open simultaneously.

EXAMPLE

```
#include <stdio.h>

#define LENGTH 80

char data[LENGTH + 2];

FILE *ff, *nf;
main()
{

      /* Open FILE1 to read. */
   ff = fopen("tso:FILE1", "r");
```

fclose Close a File
(continued)

```
      /* Open NEXTFILE to write. */
   nf = fopen("tso:NEXTFILE", "w");

      /* Read a maximum of 81 characters into the string, data. */
   while (fgets(data, LENGTH + 2, ff))
      fputs(data, nf);             /* Write data into NEXTFILE. */

   fclose(ff);
   fclose(nf);
}
```

RELATED FUNCTIONS

afflush, **close**, **fflush**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fcntl Control Open File Descriptors

SYNOPSIS

```
#include <sys/types.h>
#include <fcntl.h>

int fcntl(int filedes, int action, argument);
```

DESCRIPTION

fcntl controls open OpenEdition file descriptors and sockets. **filedes** is the file descriptor. **action** is the action to be performed on the file or socket. For a nonintegrated socket, the only actions that can be specified are **F_GETFL** and **F_SETFL**. The third argument required by some actions is **argument**. The type of this argument depends on the action.

You can specify the following actions with **fcntl**:

F_DUPFD	duplicates the file descriptor and returns the lowest available file descriptor greater than or equal to **argument**. This duplicate file descriptor refers to the same file as **filedes**.
F_DUPFD2	duplicates the file descriptor and returns a file descriptor specified by **argument**. The file descriptor specified by **argument** is closed and is then used as the new file descriptor. This duplicate file descriptor refers to the same file as **filedes**.
F_GETFD	returns the file descriptor flags for **filedes**. This action has no effect if **filedes** is a non-integrated socket.
F_SETFD	sets the file descriptor flags for **filedes**. New flag settings are specified by **argument**. This action has no effect if **filedes** is a nonintegrated socket.
F_GETFL	returns the file-status flags and file access mode flags for **filedes**. If **filedes** is a nonintegrated socket, only the setting of **O_NONBLOCK** is significant.
F_SETFL	Sets the flag status flags for **filedes**. New flag settings are specified by **argument**. **fcntl** does not change the file access mode. If **filedes** is a nonintegrated socket, only the **O_NONBLOCK** setting may be changed.
F_GETLK	returns locking information for a file.
F_SETLK	sets or clears a file segment lock.
F_SETLKW	sets or clears a file segment lock. If a lock is blocked by other locks, **fcntl** waits until it can set or clear the lock.
F_CLOSFD	closes a range of file descriptors. **argument** specifies the upper limit of the range. **filedes** is the lower limit. If **argument** is a −1, all file descriptors greater than or equal to **fildes** are closed.

The following flags and masks are defined in **<fcntl.h>**:

O_ACCMODE	This mask defines the bits that comprise the file access mode. O_ACCMODE can be ANDed with the value stored by the **F_GETFL** action to isolate the file access mode.

fcntl Control Open File Descriptors
(continued)

`FD_CLOEXEC`	If set to 1, requests that the file descriptor be closed if the process calls an **`exec`** function. If set to 0, the file descriptor remains open if the process calls **`exec`**.
`O_APPEND`	If set to 1, every write operation begins at the end of the file.
`FD_CLOFORK`	If set to 1, requests that, when a fork occurs, the file descriptor be closed for the child process. If set to 0, the file descriptor remains open for any child process.
`O_NONBLOCK`	If set to 1, read and write operations return an error if I/O cannot be immediately performed. If set to 0, read and write operations wait until I/O can be performed. Note that the traditional BSD file status flag **`FNDELAY`** is defined in **`<fcntl.h>`** as a synonym for **`O_NONBLOCK`**.
`O_RDONLY`	This file access mode value indicates the file is opened for input only.
`O_RDWR`	This file access mode value indicates the file is opened for both input and output.
`O_WRONLY`	This file access mode value indicates the file is opened for output only.

RETURN VALUE

`fcntl` returns the value specified by **`action`**. The **`fcntl`** function returns a −1 if it is not successful.

PORTABILITY

The **`F_CLOSFD`** action and the **`FD_CLOFORK`** flag are extensions defined by IBM to the POSIX.1 standard.

EXAMPLES

This example updates a counter stored in the first four bytes of an HFS file. It uses the record-locking feature of **`fcntl`** to ensure that two processes running the same program do not update the file simultaneously.

```
   /* This example requires cxompilation with the posix option. */
#include <sys/types.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

main() {
   int fd;
   long count;
   struct flock lock;
   int rc;

      /* If the file does not yet exist, create it, and treat the */
      /*  counter as 0.  If the file does exist, do not truncate  */
      /*  it, as we need the old data.                            */
```

fcntl Control Open File Descriptors
(continued)

```
   fd = open("/u/yvonne/counter", O_CREAT | O_RDWR,
             S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
   if (fd < 0) {
      perror("open error");
      exit(EXIT_FAILURE);
   }
   lock.l_type = F_WRLCK;
   lock.l_whence = SEEK_SET;
   lock.l_start = 0L;
   lock.l_len = sizeof(count);
   rc = fcntl(fd, F_SETLKW, &lock);            /* Lock bytes 0-3. */
   if (rc == -1) {
      perror("fcntl F_WRLCK error");
      exit(EXIT_FAILURE);
   }
   rc = read(fd, &count, sizeof(count));
   if (rc == -1) {
      perror("read");
      exit(EXIT_FAILURE);
   }
   if (rc < sizeof(count)) count = 0;

      /* If too few bytes read, assume count 0. */
   rc = lseek(fd, 0L, SEEK_SET);
   if (rc != 0) {
      perror("lseek error");
      exit(EXIT_FAILURE);
   }
   ++count;
   rc = write(fd, &count, sizeof(count));
   if (rc != sizeof(count)) {
      perror("write error");
      exit(EXIT_FAILURE);
   }

      /* The lock will be released when the file is closed,       */
      /*  but to be polite we will release it explicitly.         */
   lock.l_type = F_UNLCK;
   rc = fcntl(fd, F_SETLK, &lock);           /* Unlock bytes 0-3. */
   if (rc == -1) {
      perror("fcntl F_UNLCK error");
      exit(EXIT_FAILURE);
   }
   fclose(fd);
   exit(EXIT_SUCCESS);
}
```

fcntl Control Open File Descriptors
(continued)

SEE ALSO

Chapter 15, "The BSD UNIX Socket Library," in *SAS/C Library Reference, Volume 2*

Chapter 16, "Porting UNIX Socket Applications to the SAS/C Environment" in *SAS/C Library Reference, Volume 2*

"I/O Functions" on page 2-13

_fcntl Control Open HFS File Descriptors

DESCRIPTION

`_fcntl` is a version of `fcntl` designed to operate only on OpenEdition files. `_fcntl` runs faster and calls fewer other library routines than `fcntl`. The `_fcntl` function is used exactly like the standard `fcntl` function. The first argument to `_fcntl` must be the file descriptor for an open HFS file. Refer to `fcntl` for a full description.

fdopen Associate Standard I/O File with a File Descriptor

SYNOPSIS

```
#include <lcio.h>

FILE *fdopen(int fileDescriptor, const char *options)
```

The synopsis for the POSIX implementation is

```
#include <stdio.h>

FILE *fdopen(int fileDescriptor, const char *options)
```

DESCRIPTION

fdopen associates an open OpenEdition file descriptor with a pointer to a FILE structure enabling access to the file using standard I/O. This pointer enables you to control buffering and to format input and output.

fdopen accepts the following **options**. These specified options must not conflict with the current mode of the file descriptor:

r or rb	Open for reading
w or wb	Open for writing
a or ab	Open for appending
r+	Open for update
w+	Open for update
a+	Open for update at end of file

If the options string includes a “b,” the “b” is ignored. The file position indicator of the new pointer is the file offset of the file descriptor. **fdopen** clears the error indicator and the end-of-file indicator for the pointer. When the file pointer is closed, the file descriptor is also closed.

fdopen can also permit access to an open socket through a standard C FILE pointer.

RETURN VALUE

fdopen returns a FILE pointer to the control block for the new pointer. **fdopen** returns NULL if not successful.

EXAMPLE

This example uses **fdopen** to open an HFS file for standard I/O, only if the file already exists. This option is specified by the **O_EXCL** option bit for **open**, but there is no equivalent **fopen** functionality.

```
   /* This example must be compiled with the posix compiler option. */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
```

fdopen Associate Standard I/O File with a File Descriptor

(continued)

```
FILE *fopen_excl(char *pathname, char *openmode) {
   int open_opts = O_TRUNC | O_CREAT | O_EXCL;
   int fd;
   FILE *fileptr;

      /* Turn the fopen style openmode into open bits. */
      /*  We assume openmode is a valid open mode.     */
   if (openmode[0] == 'a') open_opts |= O_APPEND;
   if (strchr(openmode, '+')) open_opts |= O_RDWR;
   else open_opts |= O_WRONLY;
   fd = open(pathname, open_opts, S_IRUSR | S_IWUSR |
                                  S_IRGRP | S_IWGRP |
                                  S_IROTH | S_IWOTH);
   if (fd < 0) {          /* if the file wouldn't open */
      perror("open error");
      return NULL;
   }

       /* Make a FILE ptr for the fd.                  */
   fileptr = fdopen(fd, openmode);
   if (!fileptr) {
      perror("fdopen error");
      close(fd);
      return NULL;
   }
   return fileptr;
}
```

RELATED FUNCTIONS

fopen, **open**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

feof Test for End of File

ISO ANSI UNIX

SYNOPSIS

```
#include <stdio.h>

int feof(FILE *f);
```

DESCRIPTION

feof tests whether the stream associated with the **FILE** object addressed by **f** has reached end of file.

RETURN VALUE

feof returns 0 if the file is not positioned at end of file, or nonzero if the file is at end-of-file.

End of file is not detected until an attempt is made to read past end of file, and a call to **fseek** or **fgetpos** always resets the end of file flag.

IMPLEMENTATION

feof is implemented as an inline function. The function includes a test for a **NULL FILE** pointer and for a stream that failed to open. If you **#define** the symbol **_FASTIO**, either explicitly or using the compiler **define** option, an alternate function is used. This version of **feof** bypasses these error checks, so it executes faster.

EXAMPLE

Use **feof** to determine the end of a file opened for reading.

```
#include <stdio.h>

main()
{
  FILE *fp;
  char c;
  int count;

  fp = fopen("tso:MYFILE", "r");
  count = 0;
  while (!feof(fp) && !ferror(fp)) {
     c = getc(fp);
     ++count;
  }

  printf("The number of characters in the file 'MYFILE' was %d.\n",
         count-1);
}
```

RELATED FUNCTIONS

ferror

feof Test for End of File
(continued)

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1
“I/O Functions” on page 2-13

ferror Test Error Flag

SYNOPSIS

```
#include <stdio.h>

int ferror(FILE *f);
```

DESCRIPTION

ferror tests whether the error flag has been set for the **FILE** object addressed by **f**. This flag is set whenever an I/O function fails for any reason. The error flag for a file remains set after an error until it is cleared with **clearerr**.

RETURN VALUE

ferror returns 0 if the error flag is not set and a nonzero value if the error flag is set.

IMPLEMENTATION

ferror(f) (where **f** addresses a FILE object for a file closed by **fclose**) is nonzero. Other implementations may return 0 if **ferror** is used on a closed file.

EXAMPLE

ferror is illustrated in the example for **ftell**.

RELATED FUNCTIONS

clearerr, **clrerr**,**feof**

SEE ALSO

"Error handling" on page 3-62

"I/O Functions" on page 2-13

ffixed Test for Fixed-Length Records

SYNOPSIS

```
#include <lcio.h>

int ffixed(FILE *f);
```

DESCRIPTION

ffixed indicates whether an open file contains records of a fixed length.

RETURN VALUE

ffixed returns a nonzero value for a fixed-format file, or 0 for a variable-format file. VSAM KSDS and ESDS files are considered to be variable-format files, and RRDS and LDS files are considered to be fixed-format files. HFS files are always considered to be variable-format files.

EXAMPLE

For fixed-format files, the sequence number is the last eight characters of the line. For variable-format files, the sequence number is the first eight characters. This program removes the sequence number from a line read from the file input:

```
#include <lcio.h>
#include <string.h>

char *line;
FILE *input;

   /* If fixed format,             */
   /* throw away last 8 characters; */
if (ffixed(input))
   line[strlen(line) - 8 ]  = 0;
   /* else, move string 8 bytes to left. */
else
   memmove(line, line + 8, strlen(line) - 7);
```

RELATED FUNCTIONS

fattr

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fflush Flush Output Buffer

SYNOPSIS

```
#include <stdio.h>

int fflush(FILE *f);
```

DESCRIPTION

fflush flushes the output buffer for the stream associated with the **FILE** object addressed by **f**. The exact effect of **fflush** depends on the file type and stream type, as follows:

- For a terminal file, the current output buffer is immediately written to the terminal. The carriage is not returned if this is supported by the operating system and the terminal device.
- For a nonterminal file accessed as a binary stream, the current buffer is passed to the operating system for output and a record break is forced. Note that the record may not be physically written at this time. See "SAS/C I/O Questions and Answers" on page 3-87 for further discussion of this point.
- For a nonterminal file accessed as a text stream, **fflush** has no effect. Passing the output buffer to the operating system also forces a record break, which should occur only when a new-line character is written.

If you do not use **fflush**, output buffers are flushed as follows:

- For a terminal file, the output buffer is flushed when it is full, when a new-line character is written, when terminal input is requested, or when a character is written to the terminal using a different stream.
- For a nonterminal file accessed as text, the output buffer is flushed when it is full or when a new-line character (or any other control character that forces a new record) is written.
- For a nonterminal file accessed as binary, the output buffer is flushed when it is full.

The effect of **fflush** on a read-only file is undefined.

RETURN VALUE

fflush returns 0, or **EOF** if an error occurs.

PORTABILITY

fflush is portable when used to ensure that output to an interactive device is written out. The use of **fflush** on other files, such as disk files, does not guarantee immediate I/O, and using **fflush** to force record breaks is completely nonportable.

fflush Flush Output Buffer
(continued)

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   long partno;

   fputs("Enter part number:", stdout);
   fflush(stdout);     /* Force prompt to terminal. */
   scanf("%ld", &partno);  /* Read the part number. */
   ptintf("Request for part # %ld received.", partno);
}
```

RELATED FUNCTIONS

`afflush`, **`fsync`**

SEE ALSO

"Buffering, flushing, and prompting" on page 3-51

"I/O Functions" on page 2-13

fgetc Read a Character from a File

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *f);
```

DESCRIPTION

fgetc reads a single character from the stream associated with the **FILE** object addressed by **f** and returns the character.

RETURN VALUE

fgetc returns the next input character, or **EOF** if no character can be read.

IMPLEMENTATION

fgetc is implemented by an actual function call, not a built-in function, so it is slower than **getc**. (However, less code is generated.)

EXAMPLE

```
#include <stdio.h>

main()
{
   FILE *fp;
   int c,count;
   count = 0;

   fp = fopen("tso:MYFILE", "rb");          /* Open MYFILE to read. */
   while ((c = fgetc(fp)) != EOF)           /* Count characters.    */
      ++count;

   printf("Number of characters in the file "tso:MYFILE": %d", count);
   fclose(fp);
}
```

RELATED FUNCTIONS

getc, **getchar**, **ungetc**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

fgetpos Store the Current File Position

SYNOPSIS

```
#include <stdio.h>

int fgetpos(FILE *f, fpos_t *pos);
```

DESCRIPTION

fgetpos determines the current file position for the stream associated with the **FILE** object addressed by **f**, and it stores the file position in the object pointed to by **pos**. This object is of type **fpos_t**, which is defined in **stdio.h**. The stored value can be passed to the **fsetpos** function to reposition the file to its position at the time of the call to **fgetpos**.

You can use **fgetpos** with most types of files, using either text or binary access. Note that you can use it to obtain the current position for many files that **ftell** cannot process, including files accessed as a binary stream using the **''seq''** access method. See Tables 3.5 and 3.6 for file types that are not fully supported by **fgetpos**.

RETURN VALUES

If successful, **fgetpos** returns 0. If it fails, **fgetpos** returns a nonzero value and stores an appropriate error code in **errno**. See "The errno Variable" on page 1-7 for the list of **errno** values.

A program that uses the components of an **fpos_t** value is not portable.

IMPLEMENTATION

See "File positioning with fgetpos and fsetpos" on page 3-6 for information on the implementation of **fgetpos** and the structure of **fpos_t** values.

EXAMPLE

See the example for **fsetpos**.

RELATED FUNCTIONS

fsetpos, **ftell**, **kgetpos**, **lseek**

SEE ALSO

"File positioning with UNIX style I/O" on page 3-47

"I/O Functions" on page 2-13

fgets Read a String from a File

SYNOPSIS

```
#include <stdio.h>

char *fgets(char *str, int n, FILE *f);
```

DESCRIPTION

`fgets` reads a line of data or up to **`n-1`** characters (whichever is less) from the stream associated with the **`FILE`** object addressed by **`f`**, and it stores the input in the area addressed by **`str`**. The area must be large enough to contain **`n`** characters.

`str` addresses an array. For a stream accessed as text, characters are read into the array until **`n-1`** characters have been read, a complete line of data have been read, or end of file has been reached.

For a stream accessed as binary, characters are read until a physical new-line character is encountered, **`n-1`** characters have been read, or end of file has been reached.

`fgets` adds a null character ('\0') following the last character read into the area addressed by **`str`**.

RETURN VALUE

`fgets` returns **`str`** if successful. If end of file or an error occurs, **`fgets`** returns **`NULL`**.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFSIZE 10

main()
{
   char *buffer;
   int position;

   puts("Please enter a long line of input.");
   buffer = malloc(BUFSIZE);
   if (!buffer) exit(EXIT_FAILURE);
   *buffer = '\0';          /* Initially buffer is empty. */
   position = 0;            /* Read into start of buffer. */
   for (;;) {

         /* Read new data to last part of buffer.         */
      if (!fgets(buffer+position, BUFSIZE, stdin)) break;

         /* Stop reading if we've read the whole line.    */
      if (buffer[strlen(buffer)-1] == '\n') break;
```

fgets Read a String from a File
(continued)

```
            /* Make the buffer bigger so we can read again.  */
         buffer = realloc(buffer, strlen(buffer)+BUFSIZE);

         if (!buffer) exit(EXIT_FAILURE);
            /* Start reading after the last input character. */
         position = strlen(buffer);

      }
      if (ferror(stdin)) exit(EXIT_FAILURE);
      if (!*buffer)
         puts("There was no input data.");
      else
         printf("You entered a %d-character line:\n%s",
                strlen(buffer), buffer);
      free(buffer);
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

`afread`, **`gets`**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

fileno Return File Number

SYNOPSIS

```
#include <lcio.h>

int fileno(FILE *f);
```

Here is the synopsis for the POSIX implementation

```
#include <stdio.h>

int fileno(FILE *f);
```

DESCRIPTION

fileno returns the file number of the stream associated with the **FILE** object addressed by **f**. If **fileno** is called for a file open to an HFS file, it returns the OpenEdition file descriptor for the file. If **fileno** is called for a socket, it returns the simulated file-descriptor number for the socket. If **fileno** is called for a file that is not an OpenEdition file, it fails and returns a −1.

RETURN VALUE

fileno returns an integer file number. If **f** is 0 or is not associated with an open stream, the value returned by **fileno** is unpredictable. **fileno** of a stream that is not OpenEdition returns a −1.

EXAMPLE

This example illustrates truncating an HFS file accessed by standard I/O. **fileno** obtains the file number, and then **ftruncate** is called to truncate the file. **fflush** is called before truncation to ensure that any buffered data are flushed:

```
   /* This example must be compiled with the posix option. */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int stdtrunc(FILE *f, long pos) {
   int fd;
   int rc;

   /* Get POSIX file descriptor.           */
   fd = fileno(f);
   if (fd == -1) {
      fputs("Cannot truncate non-HFS file\n", stderr);
      return -1;
   }
   rc = fflush(f);
   if (rc != 0) {
      perror("fflush error");
      return -1;
   }
```

fileno Return File Number
(continued)

```
      /* Truncate file to requested position. */
      rc = ftruncate(fd, pos);
      if (rc == -1)
         perror("ftruncate error");
      return rc;
   }
```

RELATED FUNCTIONS

`fdopen`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

floor Round Down a Floating-Point Number

SYNOPSIS

```
#include <math.h>

double floor(double y);
```

DESCRIPTION

floor rounds a real number down to the next smaller integral real number.

RETURN VALUE

floor accepts a floating-point argument **y** and rounds it down to the next smaller integer expressed as a floating-point number.

IMPLEMENTATION

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

EXAMPLE

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

int moda, modb, df;
double score, rank;

main()
{
   puts("Enter two integers: ");
   scanf("%d %d", &moda, &modb);

   puts("Enter an integer divisor: ");
   scanf("%d",&df);

      /* Add the two numbers, divide by the given divisor, */
      /*  and then round up to the closest integer less    */
      /*  than the result.                                 */
   score = ( moda + modb);
   score /= df;
   rank = floor(score);

      /* Print the rounded result (its "floor"ed value).   */
   printf("The floor of (%d + %d)/ %d = %f\n",moda,modb,df,rank);
}
```

RELATED FUNCTIONS

ceil

floor Round Down a Floating-Point Number
(continued)

SEE ALSO

"Mathematical Functions" on page 2-6

fmax Find the Maximum of Two Doubles

SYNOPSIS

```
#include <lcmath.h>

double fmax(double s, double r);
```

DESCRIPTION

`fmax` finds the maximum of two double values, **`s`** and **`r`**.

RETURN VALUE

`fmax` returns a double value that represents the maximum of the two arguments.

IMPLEMENTATION

`fmax` is a built-in function.

EXAMPLE

```
#include <lcmath.h>
#include <stdio.h>

main()
{
   double num1, num2;  /* numbers to be compared            */
   double result;      /* holds the larger of num1 and num2 */

   puts("Enter num1 & num2 : ");
   scanf("%lf %lf", &num1, &num2);

   result = fmax(num1, num2);

   printf("The larger number is %f\n", result);
}
```

RELATED FUNCTIONS

`fmin`, **`max`**

SEE ALSO

"Mathematical Functions" on page 2-6

fmin Find the Minimum of Two Doubles

SYNOPSIS

```
#include <lcmath.h>

double fmin(double s, double r);
```

DESCRIPTION

fmin finds the minimum of two double values, **s** and **r**.

RETURN VALUE

fmin returns a double value that represents the minimum of the two arguments.

IMPLEMENTATION

fmin is a built-in function.

EXAMPLE

```
#include <lcmath.h>
#include <stdio.h>

main()
{
   double num1, num2;  /* numbers to be compared            */
   double result;      /* holds the smaller of num1 and num2 */

   puts("Enter num1 & num2 : ");
   scanf("%lf %lf", &num1, &num2);

   result = fmin(num1, num2);

   printf("The smaller number is %f\n", result);
}
```

RELATED FUNCTIONS

fmax, **min**

SEE ALSO

"Mathematical Functions" on page 2-6

fmod Floating-Point Conversion: Modulus

SYNOPSIS

```
#include <math.h>

double fmod(double y, double z);
```

DESCRIPTION

fmod determines the remainder when a real value **y** is divided by a real value **z** to produce an integer **i**. This function satisfies these relationships:

```
                      result x
y = i * z + x,        |x| < |y|
```

This function performs the same operation for **double** arguments as the **%** operator does for **int** arguments.

RETURN VALUE

fmod returns the remainder of the division. If **z** is 0, the value returned is 0. Otherwise, the returned value has the same sign as **y** and is less than **z**.

EXAMPLE

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

main()
{
   float dollars;
   float leftovers;

   puts("Enter number of dollars");
   scanf("%f", &dollars);

   leftovers = fmod(dollars, .05);
   printf("$ %.2f contains at least ", dollars);
   printf(" %.2f in pennies\n", leftovers);
}
```

RELATED FUNCTIONS

modf

SEE ALSO

"Mathematical Functions" on page 2-6

fnm Return Filename

SYNOPSIS

```
#include <lcio.h>

char *fnm(FILE *f);
```

DESCRIPTION

fnm returns the filename for the stream associated with the **FILE** object addressed by **f**. The filename can be saved and used later to reopen the file.

RETURN VALUE

fnm returns the name of the external file associated with **f**. If **f** does not identify an open stream, the effect of calling **fnm** is unpredictable.

EXAMPLE

This example uses **fnm** to name the file used for **stdout**:

```
#include <lcio.h>
#include <stdio.h>

main()
{
   printf("File name associated with stdout is %s\n", fnm(stdout));

      /* Try to open stdout as a binary stream. */
   if (!freopen(fnm(stdout), "wb", stdout))
      fputs("Unable to reopen stdout.\n", stderr);
   else
      puts("stdout has now been opened in binary mode. ");
}
```

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fopen Open a File

ISO ANSI UNIX

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(const char *name, const char *mode);
```

DESCRIPTION

The **fopen** function opens a file and returns a pointer to the associated **FILE** object. The **name** argument is the external name (sometimes called a *pathname*) of the file to be opened. Its form is system dependent. The **mode** argument is a string defining how the file will be used. For more information about open mode values, see "Open modes" on page 3-33.

RETURN VALUE

The **fopen** function returns a pointer to a **FILE** object associated with the named file. If the file cannot be opened, a **NULL** value is returned.

IMPLEMENTATION

At most, 256 files can be open at one time, including the three standard files.

EXAMPLE

This example opens for **a** the second time to add an additional line to the file:

```
#include <stdio.h>

main()
{
   FILE* test;
   int c;
   float a;

   test = fopen("tso:TEST", "w");
   puts("Enter maximum speed limit in miles:");
   scanf("%d", &c);
   fprintf(test, "Maximum speed limit is %d miles/hour.\n", c);
   fclose(test);
   a = 1.619 * c;
   test = fopen("tso:TEST", "a");
   fprintf(test, "\n In km/h, the maximum speed limit is %f\n", a);
   fclose(test);
}
```

RELATED FUNCTIONS

afopen, **freopen**, **open**

SEE ALSO

"Opening Files" on page 3-27

"I/O Functions" on page 2-13

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
- □ an optional `h`, `l`, 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.

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.

fprintf Write Formatted Output to a File
(continued)

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.

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 `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` 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`.

The type of conversion to be performed is specified by one of these characters:

`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 always has 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.

fprintf Write Formatted Output to a File
(continued)

`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.

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.

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.

fprintf Write Formatted Output to a File
(continued)

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” on page 3-1

“I/O Functions” on page 2-13

fputc Write a Character to a File

SYNOPSIS

```
#include <stdio.h>

int fputc(int c, FILE *f);
```

DESCRIPTION

fputc writes a single character **c** to the stream associated with the **FILE** object addressed by **f**.

RETURN VALUE

fputc returns the character written or **EOF** if an error occurs.

IMPLEMENTATION

fputc is implemented as an actual function call, not a built-in function, so it is slower than **putc**. (However, less code is generated.)

EXAMPLE

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main()
{
   int c;               /* must be int not char  */
   char filename[60];
   FILE *infile, *outfile;

   puts("Enter the name of your input file:");
   memcpy(filename, "tso:", 4);
   gets(filename+4);
   infile = fopen(filename, "r");
   if (!infile){
      puts("Failed to open input file.");
      exit(EXIT_FAILURE);
   }
   puts("Enter the name of your output file:");
   memcpy(filename, "tso:", 4);
   gets(filename+4);
   outfile = fopen(filename, "w");
   if (!outfile){
      puts("Failed to open output file.");
      exit(EXIT_FAILURE);
   }

      /* Read characters from file MYFILE.       */
   while ((c = fgetc(infile)) != EOF)
```

fputc Write a Character to a File
(continued)

```
            /* Write characters to YOURFILE.          */
         if (fputc(c, outfile) == EOF) break;
      fclose(infile);
   }
```

RELATED FUNCTIONS

`putc`, **`putchar`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fputs Write a String to a File

SYNOPSIS

```
#include <stdio.h>

int fputs(const char *str, FILE *f);
```

DESCRIPTION

`fputs` writes the characters in the string addressed by **`str`** to the stream associated with the **`FILE`** object addressed by **`f`**. Unlike **`puts`**, **`fputs`** does not write a new-line character after the string.

RETURN VALUE

`fputs` returns an unspecified value unless an error occurs, in which case it returns **`EOF`**.

EXAMPLE

```
#include <stdio.h>

#define LENGTH 80

char data[LENGTH + 2];

FILE *ff, *nf;

main()
{
      /*  Open FILE1 to read.    */
   ff = fopen("tso:FILE1", "r");

      /* Open NEXTFILE to write. */
   nf = fopen("tso:NEXTFILE", "w");

      /* Read a maximum of 81 characters into the data buffer. */
   while (fgets(data, LENGTH + 2, ff))
      fputs(data, nf);            /* Write data into NEXTFILE. */
}
```

RELATED FUNCTIONS

`puts`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fread Read Items from a File

ISO ANSI UNIX

SYNOPSIS

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t count, FILE *f);
```

DESCRIPTION

fread reads one or more items of any type from the stream associated with the **FILE** object addressed by **f**. The **size** function defines the size of each item, **count** defines the number of items to be read, and **ptr** addresses the area into which the items are to be read.

Although **fread** may be used to read characters, it is more frequently used to read noncharacter data, such as structured data. Except when **fread** is used to read printable character data, you should limit its use to binary streams because the library's transformation of control characters may change the data in unpredictable ways when reading and writing text streams.

Calls to **fread** to obtain items of type **typeval** commonly have this form:

```
typeval buf[count];
fread(buf, sizeof(typeval), count, f);
```

RETURN VALUE

fread returns the number of items successfully read. It returns 0 if no items are read because of an error or an immediate end of file.

CAUTION

When using **fread**, remember that **size** is not necessarily a multiple of the record size, and that **fread** ignores record boundaries.

DIAGNOSTICS

The return value from **fread** does not indicate whether the call is completely successful. You can use the **ferror** function to determine whether an error occurs.

If **fread** returns a value of 0, but **count** is greater than 0, an error or end of file occurred before any items were read.

Attempting to read a fraction of an item (for example, calling **fread** with a size of 4 when the file contains three characters) is an error.

PORTABILITY

Some non-ISO/ANSI C implementations may return 0 in case of error, even though some items are successfully read.

IMPLEMENTATION

If **count** is less than one, no input takes place. If an error occurs during the input operation, the file position is unpredictable.

fread
(continued)

Read Items from a File

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   FILE *inf, *outf;
   int data[40];
   size_t count;

   if ( (inf=fopen("tso:READ", "r")) == NULL ) {
      fprintf(stderr, "Can't open READ.\n");
      exit(1);
   }

   if ( (outf=fopen("tso:WRITE", "w")) == NULL ) {
      fprintf(stderr, "Can't open WRITE.\n");
      exit(1);
   }

   while ( !ferror(inf) && !ferror(outf) ) {

         /* Test for error. Read items from READ and store */
         /*  the number of items read in count.            */
      count = fread(data, sizeof(data[0]), 40, inf);

      if (count == 0)
         break;

         /* Write items to WRITE and store the number of   */
         /*  items written into count.                     */
      count = fwrite((void*)data, sizeof(data[0]), count, outf);

      if (count < 40)
         break;
      }
   fclose(inf);
   fclose(outf);
}
```

RELATED FUNCTIONS

afread

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

free Free a Block of Memory

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

void free(void *block);
```

DESCRIPTION

`free` frees a block of memory previously allocated by **`malloc`** or **`calloc`**. **`block`** is a pointer to the memory block.

RETURN VALUE

`free` 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

If an entire page of memory is unused after a **`free`** call, the page is returned to the operating system unless the page is included in the initial heap area whose size can be specified with a run-time argument.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct LIST
{
   struct LIST *next;
   char text[0];  /* Zero-length arrays are a SAS/C extension. */
};

main(int argc, char *argv[])
{
   struct LIST *p;
   struct LIST *q;
   struct LIST list;
   char str[256];
   int size;
```

free Free a Block of Memory
(continued)

```
      while (1){
         puts("\nBegin new group...");
         for (q = &list; ; q = p){
            puts("Enter a text string: ");
            if (fgets(str,sizeof(str),stdin) == NULL){
               break;
            }
            if (str[0] == '\0'){
               if (q == &list)
                  exit(EXIT_SUCCESS);
               break;
            }
            size = sizeof(struct LIST) + strlen(str) + 1;
            p = (struct LIST *)malloc(size);
            if (p == NULL){
               puts("No more memory");
               exit(EXIT_FAILURE);
            }
            q->next = p;
            p->next = NULL;
            strcpy(p->text, str);
         }

         puts("\n\nTEXT LIST...");

            /* Be sure to copy the next pointer from */
            /* the current block before you free it. */

         p = list.next;
         while(p != NULL){
            q = p->next;
            printf(p->text);
            free((char *)p);
            p = q;
         }
         list.next = NULL;

         exit(EXIT_SUCCESS);
      }
   }
```

RELATED FUNCTIONS

pfree

SEE ALSO

"Memory Allocation Functions" on page 2-10

freopen Reopen a File

ISO ANSI UNIX

SYNOPSIS

```
#include <stdio.h>

FILE *freopen(const char *name, const char *mode, FILE *oldf);
```

DESCRIPTION

The **freopen** function closes the stream associated with the **FILE** object addressed by **oldf** and then reopens it, using the filename and open mode specified by **name** and **mode**.

name is the external name of the file to be opened. The form of **name** is system dependent. Note that the **name** to be opened may be different from the filename currently associated with **oldf**. The **mode** string defines how the file is used. For more information about open-mode values, see "Open modes" on page 3-33.

Portable use of **freopen** requires that **oldf** identify an open file. This implementation permits **oldf** to reference a closed file, which permits you to reuse a **FILE** pointer by calling **freopen** after **fclose**.

RETURN VALUE

If **freopen** is successful, the value of **oldf** is returned. The **FILE** object addressed by **oldf** is now associated with the file specified by **name**.

If **freopen** is unsuccessful, a **NULL FILE** pointer is returned. Further use of **oldf** after an unsuccessful **freopen** is not permitted.

EXAMPLE

This example uses **freopen** to change the **stderr** file, thereby enabling library diagnostic messages to be redirected:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

main()
{
   FILE *f;
   char answer;

   puts("Do you wish to save library messages on disk? (Y or N)?");
   answer = toupper(getchar());
   if (answer == 'Y'){
         f = freopen("tso:saved.messages", "w", stderr);
         if (!f){
               puts("Failed to reopen stderr.");
               exit(EXIT_FAILURE);
         }
   }
```

freopen Reopen a File
(continued)

```
            /* The following fopen() is deliberately invalid and causes */
            /*  a library diagnostic to be written to stderr.           */
         f = fopen("tso:impossible.file.name", "w");
         exit(EXIT_SUCCESS);
      }
```

RELATED FUNCTIONS

afreopen, **fopen**

SEE ALSO

"Opening Files" on page 3-27

"I/O Functions" on page 2-13

frexp Floating-Point Conversion: Fraction-Exponent Split

SYNOPSIS

```
#include <math.h>

double frexp(double y, int *j);
```

DESCRIPTION

`frexp` separates a floating-point argument **`y`** into a fraction and an integer exponent of 2.

RETURN VALUE

`frexp` returns a real number that is equal to or greater than .5 and less than 1.0. The exponent of 2 is stored in the location addressed by **`j`**.

USAGE NOTES

`frexp` is useful in situations that require repeated multiplication by 2. If the next multiplication causes an overflow or underflow, you can use **`frexp`** to separate the mantissa from the exponent. This gives you complete control over the exponent and mantissa so you can operate on them separately without any loss of precision.

RELATED FUNCTIONS

`ldexp`

SEE ALSO

"Mathematical Functions" on page 2-6

fscanf Read Formatted Input from a File

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][h | l | L]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.
- □ An optional letter has the following meanings:
  - □ 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`** before a **`d`**, **`i`**, or **`n`** conversion specifier indicates that the corresponding argument is a pointer to **`long int`** instead of **`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`** 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 **`l`** before an **`e`**, **`f`**, or **`g`** conversion specifier indicates that the corresponding argument is a pointer to **`double`** instead of **`float`**.
  - □ An **`L`** before an **`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:

`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.

fscanf Read Formatted Input from a File
(continued)

`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 *`.
`e`, `E`, `f`, `g`, or `G`	matches a floating-point number. The corresponding argument should be `float *`.
`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.

fscanf Read Formatted Input from a File
(continued)

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;
   float amount;
   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);
```

fscanf Read Formatted Input from a File

(continued)

```
      for(;;){
         count = fscanf(tmpf, "%f %s", &amount, unit);
         if (feof(tmpf)) break;
         if (count < 2){
            puts("Unexpected error in input data.");
            exit(EXIT_FAILURE);
         }
            printf("amount = %f, units = \\"%s\\"\\n", amount, unit);
      }
      fclose(tmpf);
   }
```

RELATED FUNCTIONS

fprintf, **scanf**, **sscanf**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

fseek Reposition a File

SYNOPSIS

```
#include <stdio.h>

int fseek (FILE *f, long int offset, int type);
```

DESCRIPTION

`fseek` repositions the stream associated with the `FILE` object pointed to by `f`, as specified by the values of `offset` and `type`. The `type` value must be one of `SEEK_SET`, `SEEK_CUR`, or `SEEK_END`. (These constants are defined in `<stdio.h>`.) Each of the `type` values refers to a specific location in the file, as follows:

- `SEEK_SET` refers to the start of the file.
- `SEEK_CUR` refers to the current file position.
- `SEEK_END` refers to the end of file.

The interpretation of the `offset` value depends on the value of `type` and whether `f` identifies a text or binary stream.

For a binary stream, the `offset` value specifies the offset in characters of the new position from the location identified by `type`. Because of this ISO/ANSI requirement, `fseek` can be called for a binary stream only when the `''rel''` access method is used. Note that the `offset` value may be either positive or negative, but positioning before the start of the file is not supported. See "File positioning with fseek and ftell" on page 3-5 for the details of positioning beyond the end of file.

When `fseek` is used with a text stream, two cases can be distinguished. If the value of `offset` is 0, the file is repositioned to the location identified by `type`. An `offset` value other than 0 is supported for a text stream only if `type` is `SEEK_SET`. In this case, `offset` must be a value previously returned by `ftell` for the same stream, and `fseek` restores the file position at the time of that call. Note that, in this case, the value in `offset` is an internal representation of the file position and cannot be interpreted as a relative character number.

After a call to `fseek` on a stream that permits both reading and writing, the next file operation may be input or output. (However, for an MVS PDS member, you can switch from reading to writing only at the start of the file.)

RETURN VALUE

If successful, `fseek` returns 0. If it fails, `fseek` returns a nonzero value and stores an appropriate value in `errno`. See the list of `errno` values in "The errno Variable" on page 1-7 .

CAUTIONS

If output occurs after a call to `fseek` on a text stream, characters following the point of output may be erased from the file. This occurs when `trunc=yes` is in effect. Therefore, when `trunc=yes` is in effect, you are unable to return to a previous file position if output has been performed before that point.

See "Opening Files" on page 3-27 for more information on the `trunc` amparm.

fseek Reposition a File

(continued)

PORTABILITY

Be cautious when porting an **`fseek`** application because the implementation of **`fseek`** varies from system to system and library to library.

IMPLEMENTATION

Refer to “File positioning with fgetpos and fsetpos” on page 3-6 for implementation details.

EXAMPLE

See the example for **`ftell`**.

RELATED FUNCTIONS

`ksearch`, **`kseek`**, **`lseek`**, **`fsetpos`**, **`ftell`**

SEE ALSO

“File positioning with UNIX style I/O” on page 3-47

“I/O Functions” on page 2-13

fsetpos Reposition a File

SYNOPSIS

```
#include <stdio.h>

int fsetpos(FILE *f, const fpos_t pos);
```

DESCRIPTION

`fsetpos` positions the stream associated with the `FILE` object addressed by `f` to the position specified by the object pointed to by `pos`. This object is of type `fpos_t`, which is defined in `stdio.h`.

The value of the object addressed by `pos` may have been stored by a previous call to `fgetpos` for the same stream, or it may have been constructed by some other mechanism. The use of values that have not been generated by `fgetpos` is nonportable. See "File positioning with fgetpos and fsetpos" on page 3-6 for information on the interpretation of file-position values.

The `fsetpos` function can be used with most files, accessed either as text or binary. Note that it may be used to reposition files that `fseek` cannot process, including files accessed as a binary stream using the `''seq''` access method. See Tables 3.5 and 3.6 for file types that do not fully support `fsetpos`.

After a call to `fsetpos` on a stream that permits both reading and writing, the next file operation may be input or output.

RETURN VALUE

If successful, `fsetpos` returns 0. If it fails, `fsetpos` returns a nonzero value and stores an appropriate error code in `errno`. See the list of `errno` values in "The errno Variable" on page 1-7 . Calls to `fsetpos` with an invalid `pos` may not be detected immediately, but such calls will probably cause errors whenever the file is next read or written.

CAUTIONS

If output occurs after a call to `fsetpos`, characters following the point of output may be erased from the file. This occurs when `trunc=yes` is in effect. Therefore, when `trunc=yes` is in effect, you are unable to return to a previous file position if output has been performed before that point.

See "Opening Files" on page 3-27 for more information on the `trunc` amparm.

A program that makes direct use of the components of an `fpos_t` value is not portable.

IMPLEMENTATION

See "File positioning with fgetpos and fsetpos" on page 3-6 for information on the implementation of `fgetpos` and the structure of `fpos_t` values.

fsetpos Reposition a File
(continued)

EXAMPLE

This example illustrates how to use **`fsetpos`** and **`fgetpos`** to build and use an index file for access to individual lines of a file:

```
#include <stdio.h>
#include <string.h>
#define KEYLEN 10              /* size of key in record      */
#define DATALEN 70             /* size of data area in record */
#define TABSIZE 100            /* maximum number of records   */

struct  {
   char keyval[KEYLEN];
   fpos_t location;
   }  keytable[TABSIZE];

struct record {
   char keyval[KEYLEN];
   char data[DATALEN];
   };

int filesize;

void bldtable(FILE *fileptr);
int findrec(FILE *fileptr, char keyval[KEYLEN], struct record *input);

main()
{
   FILE *fileptr;
   struct record output;
   char key[KEYLEN] = "PAR-94412M";      /* key to be found */

      /* Open data file and build index file.              */
      /* Example of information in data file:              */

      /* PAR-97612MPearl & Black            $325.00        */
      /* PAR-94412MMarbled Green            $275.00        */

   if ((fileptr = fopen("ddn:DATA", "rb")) != NULL)
     bldtable(fileptr);
   else{
     puts("Unable to open input file.");
     exit(99);
   }

      /* Find desired key.                                 */
   if ( !findrec(fileptr, key, &output) )
     printf("Data area associated with key %.*s is: %.*s\n",
             KEYLEN, key, DATALEN, output.data);
   else
     puts("Unable to find matching key.");
   }
```

fsetpos Reposition a File
(continued)

```
   /* Build the table for key and record addresses.         */
void bldtable(FILE *fileptr)
{
   struct record input;
   int index = 0;

   for (;;){

         /* Store file pointer location.                     */
      fgetpos(fileptr, &keytable[index].location);

         /* Read one record.                                 */
      fread(&input, sizeof(struct record), 1, fileptr);
      if (feof(fileptr) || ferror(fileptr))
         break;

         /* Save the keyval.                                 */
      memcpy(keytable[index].keyval, input.keyval, KEYLEN);
      ++index;
   }

   filesize = index;
   return;
}

   /* Find the key.                                          */
int findrec(FILE *fileptr, char keyval[KEYLEN],
    struct record *input)
{
   int index;

      /* Search keytable for specified keyval.               */
   for(index = 0; index < filesize; ++index)
   if (memcmp(keyval, keytable[index].keyval,KEYLEN) == 0)
      break;

      /* Was the key found?                                  */
   if (index >= filesize)
      return -1;

      /* If found, read complete record from file.           */
   fsetpos(fileptr, &keytable[index].location);
   fread(input, sizeof(struct record), 1, fileptr);
   return 0;
```

RELATED FUNCTIONS

`fgetpos`, **`fseek`**, **`lseek`**

fsetpos Reposition a File

(continued)

SEE ALSO

"File positioning with UNIX style I/O" on page 3-47

"I/O Functions" on page 2-13

fstat Determine File Status by Descriptor

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int fstat(int filedes, struct stat *info);
```

DESCRIPTION

fstat gets status information for an HFS file and returns it in a **stat** structure, defined in **<sys/stat.h>**. The **filedes** file descriptor is the file descriptor for which status information is needed. **filedes** must be an open file descriptor associated with an OpenEdition HFS file. **info** is the area of memory in which the status information is stored.

The following macros (defined in **<stat.h>**) are available:

S_ISBLK(mode)	for block special files
S_ISCHR(mode)	for character special files
S_ISDIR(mode)	for directories
S_ISFIFO(mode)	for pipes and FIFO special files
S_ISLNK(mode)	for symbolic links
S_ISREG(mode)	for regular files
S_ISSOCK(mode)	for integrated sockets

RETURN VALUE

fstat returns a 0 if it is successful and a −1 if it is not successful.

EXAMPLE

The following example illustrates a function that is called to determine whether two file descriptors access the same file. Note that in this example two different HFS files must either have different device numbers or different inode numbers:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int samefile(int fd1, int fd2) {
   struct stat stat1, stat2;
   int rc;
   rc = fstat(fd1, &stat1);
   if (rc == -1) {
      perror("fstat error");
      return -1;
   }
```

fstat Determine File Status by Descriptor

(continued)

```
      rc = fstat(fd2, &stat2);
      if (rc == -1) {
         perror("fstat error");
         return -1;
      }
      if (stat1.st_dev == stat2.st_dev && stat1.st_ino == stat2.st_ino)
         return 1;
      else return 0;
   }
```

RELATED FUNCTIONS

`fattr`, **`lstat`**, **`stat`**

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

fsync Flush UNIX style File Buffers to Disk

SYNOPSIS

```
#include <fcntl.h>

int fsync(int fn);
```

The synopsis for the POSIX implementation is

```
#include <unistd.h>

int fsync(int fn);
```

You may use either header file in your program.

DESCRIPTION

The **fsync** function flushes the output buffers to disk for the UNIX style file whose file number is **fn**. The **fsync** function also performs additional system-dependent operations to ensure that the data are accessible later, even if the program or the system fails later. The file position is unchanged. **fsync** returns when the output buffers are flushed or when an error is detected.

RETURN VALUES

The **fsync** function returns 0, or **EOF** if an error occurs.

CAUTIONS

Using **fsync** is expensive for files that are processed using a temporary copy because the file's entire contents must be copied each time **fsync** is called.

EXAMPLE

```
#include <fcntl.h>

extern int num_updates;
extern int fd;
int rc;

if (num_updates == 100) {
   rc = fsync(fd);            /* Flush updates to disk. */
   if (rc != 0) {
      puts("Error saving recent updates.");
      close(fd);
      abort();
   }
   num_updates = 0;           /* Reset update counter.  */
}
```

RELATED FUNCTIONS

afflush

fsync Flush UNIX style File Buffers to Disk
(continued)

SEE ALSO

"I/O Functions" on page 2-13

_fsync Flush HFS File Buffers to Disk

DESCRIPTION

`_fsync` is a version of `fsync` designed to operate only on HFS files. `_fsync` runs faster and calls fewer other library routines than `fsync`. The `_fsync` function is used exactly like the standard `fsync` function. The argument to `_fsync` must be the file descriptor for an open HFS file. See `fsync` for a full description.

ftell Obtain the Current File Position

SYNOPSIS

```
#include <stdio.h>

long int ftell(FILE *f);
```

DESCRIPTION

ftell returns the current position of the stream associated with the **FILE** object addressed by **f**, expressed as a **long** integer. The value returned by **ftell** can be passed later to a call to the **fseek** function for the same stream to restore the position at the time of the call to **ftell**.

When **ftell** is called for a binary stream, the position is expressed as the number of bytes from the start of the file. **ftell** is supported for a binary stream only when the **''rel''** access method is used or for an HFS file.

When **ftell** is called for a text stream, the value returned is an encoded form of the file position as stored by the **fgetpos** function. For a text stream, the difference between two file positions may not be related to the amount of separation between the two positions.

See “File positioning with fseek and ftell” on page 3-5 for more information on the implementation of **ftell**. See Tables 3.5 and 3.6 for file types for which **ftell** is not fully implemented.

RETURN VALUE

ftell returns the current file position expressed as a **long int**, if possible. If **ftell** is unable to successfully determine the file position, or if it cannot be stored in a **long int**, **-1L** is returned. In the latter case, an appropriate value is also stored in **errno**. See the list of **errno** values in “The errno Variable” on page 1-7 .

PORTABILITY

Be cautious when porting an **ftell** application because the implementation of **ftell** varies from system to system and library to library.

IMPLEMENTATION

Refer to “File positioning with fseek and ftell” on page 3-5 for implementation details.

EXAMPLE

This example counts the number of desired items in the file after the current position. Then it allocates storage, restores the file position, and reads the items:

```
#include <stdio.h>

main()
{
struct item {
   char name[40];
```

ftell Obtain the Current File Position
(continued)

```
   int age;
   };

struct item new_item;
struct item *all_items;
long start;                       /* file position pointer      */

FILE *f;
int i, count=0;

f = fopen("tso:ITEMFILE", "rb");
start = ftell(f);                 /* Set file position pointer. */

   /* Count the number of items in the file with age over 20.   */
while(!feof(f) && !ferror(f)) {
   if (fread(&new_item, sizeof(new_item), 1, f))
      if (new_item.age > 20)
         ++count;
   if (ferror(f)) {
      puts("Error while reading file..exiting.");
      exit(1);
   }
}

   /* Seek to START location. */
if (fseek(f, start, SEEK_SET)) {
   puts("Could not locate start of file.");
   exit(1);
}

   /* Allocate space for all items. */
all_items = (struct item*)calloc(count, sizeof(struct item));
if (!all_items)                   /* Was memory allocated?      */
   exit(1);

   /* Read in items and store only the ones with age > 20.      */
i = 0;
while (!feof(f) && i < count)
   if (fread((all_items+i), sizeof(struct item), 1, f))
      if (all_items[i].age > 20)
         i++;
}
```

RELATED FUNCTIONS

fseek, **fsetpos**, **lseek**

ftell Obtain the Current File Position
(continued)

SEE ALSO

"File positioning with standard I/O (fseek and ftell)" on page 3-49

"I/O Functions" on page 2-13

fterm Terminal File Test

SYNOPSIS

```
#include <lcio.h>

int fterm(FILE *f);
```

DESCRIPTION

`fterm` returns an indication of whether or not the stream associated with the **`FILE`** object addressed by **`f`** is assigned to an interactive terminal.

`fattr` contains the functionality of **`fterm`** in addition to other capabilities and is a better choice in most cases.

RETURN VALUE

`fterm` returns 0 for a nonterminal **`FILE`**. The **`fterm`** function returns a nonzero value if the **`FILE`** is assigned to a TSO, CMS, or OpenEdition terminal.

EXAMPLE

```
#include <lcio.h>
#include <stdio.h>

main()
{
   int k;
   k = fterm(stdout);

   if (k == 0)
      puts("File 'stdout' is not assigned to an interactive "
              "terminal.");
   else
      puts("File 'stdout' is assigned to an interactive terminal.");
}
```

RELATED FUNCTIONS

`fattr`, **`fopen`**, **`isatty`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

ftruncate Truncate a File

POSIX

SYNOPSIS

```
#include <unistd.h>

int ftruncate(int fn, off_t length);
```

DESCRIPTION

ftruncate truncates an OpenEdition file open for write access to **length** in bytes. **fn** is the file descriptor for the file to be truncated. **ftruncate** does not change the current file position.

RETURN VALUE

ftruncate returns a 0 if it is successful and a −1 if it is not successful.

IMPLEMENTATION

Data beyond the end of the truncated file are lost. If the file size is smaller than the specified **length**, the file is unchanged.

EXAMPLE

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

main()
{
   FILE *testfile;
   int fd, long size;
   char flname[] ="trunc.test";

   if ((fd=creat(flname,S_IRUSR1)) < 0)
      perror("creat() error");
   else {
      if ((testfile = fdopen(fd, "w")) == NULL) {
         perror("fdopen() error");
         close(fd);
      }
   }
   fputs("ABCDEFGHIJKLMNOPQRSTUVWXYZ",testfile);
   fflush(testfile);
   size = lseek(fd, 0, SEEK_END);
   printf("The file is %ld bytes long.\n", size);

   if (ftruncate(fd, 3) !=0)
      perror("ftruncate() error");
```

ftruncate Truncate a File

(continued)

```
      else {
         size = lseek(fd, 0, SEEK_END);
         printf("The file is now %ld bytes long.\n", size);
      }
   fclose(testfile);
   unlink(flname);
   }
```

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

fwrite Write Items to a File

SYNOPSIS

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size,
              size_t count, FILE *f);
```

DESCRIPTION

fwrite writes one or more items of any type to the stream associated with the **FILE** object addressed by **f**. The size of each item is defined by **size**, **count** defines the number of items to be written, and **ptr** addresses the area containing the items to be written.

Although **fwrite** may be used to write characters, it is more frequently used to write noncharacter data, such as structured data. Except when **fwrite** is used to write printable character data, you should limit its use to binary streams because the library's transformation of control characters may change the data in unpredictable ways when reading and writing text streams.

Calls to **fwrite** to write items of type **typeval** commonly have the form

```
typeval buf[count];
fwrite(buf, sizeof(typeval), count, f);
```

RETURN VALUE

fwrite returns the number of items successfully written. It returns 0 if no items are written because of an error.

CAUTION

When using **fwrite**, remember that **size** is not necessarily a multiple of the record size, and that **fwrite** ignores record boundaries.

DIAGNOSTICS

If **fwrite** returns a value of 0, but **count** is greater than 0, an error occurred before any items were written.

PORTABILITY

Some implementations may return 0 in case of error, even though one or more items were successfully written.

IMPLEMENTATION

If **count** is less than 1, no output takes place. If an error occurs during the output operation, the file position is unpredictable.

EXAMPLE

The use of **fwrite** is illustrated in the example for **fread**.

RELATED FUNCTIONS

afwrite

fwrite Write Items to a File
(continued)

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

gamma Compute the Logarithm of the Gamma Function

UNIX

SYNOPSIS

```
#include <lcmath.h>

double gamma(double x);
```

DESCRIPTION

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

$$gamma(x) = log\left(\int_0^\infty t^{x-1} e^{-t} dt\right)$$

RETURN VALUE

gamma returns the logarithm of the gamma function of its argument.

DIAGNOSTICS

The run-time library writes an error message to the standard error file (**stderr**) if **x** is a negative number or 0. In this case, the function returns **HUGE_VAL**, the largest positive floating-point number that can be represented. Also, the run-time library writes an error message to the standard error file (**stderr**) if **x** is greater than 0.42686124520937873e74 in scientific notation. In this case, the function returns **HUGE_VAL**, the largest positive floating-point number that can be represented.

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

PORTABILITY

gamma is portable to many, but not all, non UNIX C implementations.

EXAMPLE

This example computes the logarithm of the gamma function using **gamma**:

```
#include <stdio.h>
#include <lcmath.h>
#define SVECTOR .7854

main()
{
   double lgamv;
   lgamv = gamma(SVECTOR);
   printf("gamma(%f) = %f\n", SVECTOR, lgamv);
}
```

RELATED FUNCTIONS

_matherr

gamma Compute the Logarithm of the Gamma Function
(continued)

SEE ALSO

"Mathematical Functions" on page 2-6

getc Read a Character from a File

SYNOPSIS

```
#include <stdio.h>

int getc(FILE *f);
```

DESCRIPTION

getc reads a single character from the stream associated with the **FILE** object addressed by **f**.

RETURN VALUE

getc returns the next input character or **EOF** if no character is read. A return value of **EOF** indicates that the end of file has been reached or that an error has occurred. The **ferror** function can be called to distinguish these cases.

IMPLEMENTATION

getc is implemented as a built-in function. A subroutine call is executed only if no characters remain in the current input buffer.

The code generated for **getc** normally includes tests for a 0 **FILE** pointer and for a stream that failed to open. If you define the symbol **_FASTIO** using **#define** or the **define** compiler option before including **<stdio.h>**, no code is generated for these checks. This enables you to improve the performance of debugged programs that use **getc**.

EXAMPLE

This example copies one file to another, translating all uppercase characters to lowercase characters, and all lowercase characters to uppercase characters.

```
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define _FASTIO        /* Improve getc/putc performance. */

main()
{
   int ch;
   FILE *in, *out;
   char filename[60];

   puts("Enter the name of your input file:");
   memcpy(filename, "cms:", 4);
   gets(filename+4);
   in = fopen(filename, "r");
   if (!in){
      puts("Failed to open input file.");
      exit(EXIT_FAILURE);
   }
```

getc Read a Character from a File
(continued)

```
      puts("Enter the name of your output file:");
      memcpy(filename, "cms:", 4);
      gets(filename+4);
      out = fopen(filename, "w");
      if (!out){
         puts("Failed to open output file.");
         exit(EXIT_FAILURE);
      }

      for(;;){
         ch = getc(in);
         if (ch == EOF) break;
         if (islower(ch)) putc(toupper(ch), out);
         else putc(tolower(ch), out);
         if (ferror(out)) break;
      }

      if (ferror(in) || ferror(out))    /* Check for error. */
         exit(EXIT_FAILURE);
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

fgetc, **getchar**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

getchar Read a Character from the Standard Input Stream

SYNOPSIS

```
#include <stdio.h>

int getchar(void);
```

DESCRIPTION

getchar reads a character from the stream **stdin**.

RETURN VALUE

getchar returns the next input character or **EOF** if no character can be read.

IMPLEMENTATION

getchar is a macro that expands into **getc(stdin)**.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   char filename[60];
   int words = 0, lines = 0, bytes = 0;
   int spacing = 1;
   int ch;

   puts("Enter the input file name:");
   memcpy(filename, "tso:", 4);
   gets(filename+4);
   if (freopen(filename, "r", stdin) == 0){
      puts("File could not be opened.");
      exit(EXIT_FAILURE);
   }

      /* Read the file and count bytes, lines, and words. */
   for(;;){
      ch = getchar();
      if (ch == EOF) break;
      ++bytes;
      if (ch == '\n') ++lines;

         /* If the input character is a nonspace character */
         /* after a space character, start a new word.     */
      if (isspace(ch)) spacing = 1;
      else if(spacing){
         spacing = 0;
         ++words;
      }
   }
```

getchar *(continued)*

Read a Character from the Standard Input Stream

```
      printf("The input file contains %d bytes\n", bytes);
      printf("%d lines and %d words.\n", lines, words);
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

getc

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

getcwd Determine Working Directory Pathname

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char *buffer, int size);
```

DESCRIPTION

getcwd finds the pathname of the working OpenEdition directory. **buffer** is the buffer in which this information is stored. The size of **buffer** must be large enough to hold the pathname and a terminating null. **size** is the number of characters in the buffer area.

RETURN VALUE

getcwd returns a pointer to the buffer if successful and a NULL pointer if not successful.

IMPLEMENTATION

When **getcwd** is called in a program that is not compiled under POSIX, the returned directory name includes an **''hfs:''** style prefix.

EXAMPLE

See the example for **chdir**.

RELATED FUNCTIONS

chdir

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

getenv Get Value of Environment Variable

SYNOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

DESCRIPTION

The **getenv** function searches an environment-variable list for the string **name** and returns a corresponding value. The variable name may be translated to uppercase letters, depending on the operating environment, as described in Chapter 4, "Environment Variables" on page 4-1. In some contexts, environment-variable names are limited to about 250 characters.

Depending on the environment, if **name** contains a period, the portion of the **name** preceding the period is interpreted as a group name, as described in Chapter 4, "Environment Variables" on page 4-1. Group names are limited to 8 characters.

See Chapter 4, "Compiling C Programs" and Chapter 8, "Run-Time Argument Processing," in the *SAS/C Compiler and Library User's Guide* for information on defining environment variables.

RETURN VALUE

getenv returns a pointer to the environment-variable value if **name** was found. This pointer may address a static buffer, which is reused by the next call to **getenv**. If **name** is not found in the environment-variable list, **getenv** returns **NULL**.

CAUTIONS

Environment-variable values are not altered by the C library. However, environment variables specified on the TSO or CMS command line may be converted to uppercase letters by TSO or CMS before control is given to the C program.

PORTABILITY

getenv compiles with the POSIX.1 and POSIX.1a standards for C programs invoked by an **exec** function.

EXAMPLE

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

main()
{
   char *locale_string;
   locale_string = getenv("_LOCALE");
```

getenv Get Value of Environment Variable

(continued)

```
      if (locale_string)
         printf("The current default locale is %s\n", locale_string);
      else puts("The _LOCALE environment variable is not set.");
   }
```

RELATED FUNCTIONS

`clearenv`, **`execshv`**, **`putenv`**, **`setenv`**

SEE ALSO

Chapter 6, “Executing C Programs,” in *SAS/C Compiler and Library User’s Guide*

Chapter 4, “Environment Variables” on page 4-1

“System Interface and Environment Variables” on page 2-17

getlogin Determine User Login Name

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

char *getlogin(void);
```

DESCRIPTION

getlogin returns the login name for the current process. Under OpenEdition MVS, this is the same as the userid defined for the batch job or TSO session that generated the process. The name string is stored in a static area and may be overwritten by subsequent calls to **getlogin**. The **getlogin** function fails if OpenEdition is not active or installed.

RETURN VALUE

getlogin returns a pointer to the name string if successful, and a NULL pointer if unsuccessful.

EXAMPLE

This example tests to see if the effective user ID is the ID belonging to the login name.

```
#include <sys/types.h>
#include <unistd.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>

main() {
   char *name;
   struct passwd *unifo;

   name = getlogin();
   if (!name) {
      perror("getlogin failure");
      exit(EXIT_FAILURE);
   }
   unifo = getpwnam(name);
   if (!unifo) {
      perror("getpwnam failure");
      exit(EXIT_FAILURE);
   }
   if (unifo->pw_uid == geteuid())
      puts("Your user ID number matches the effective user ID.");
   else
      puts("Your user ID does not match the effective user ID.");
   exit(EXIT_SUCCESS);
}
```

getlogin *(continued)* Determine User Login Name

RELATED FUNCTIONS

`cuserid`

SEE ALSO

Chapter 4, "Environment Variables" on page 4-1

"System Interface and Environment Variables" on page 2-17

gets Read a String from the Standard Input Stream

SYNOPSIS

```
#include <stdio.h>

char *gets(char *str);
```

DESCRIPTION

gets reads a line of data from the stream **stdin** and stores the data in the area addressed by **str**. The **gets** function terminates the input line with a null character.

Note: Although many of the code examples in this book use **gets**, this function should not be used in production code. **gets** was used to keep the SAS/C examples as simple as possible.

RETURN VALUE

gets returns **str**. If no characters are read due to end of file or if an error occurs during the read, 0 is returned.

IMPLEMENTATION

gets(str) is approximately equivalent to the following, except that the new-line character that terminates the input line is not stored:

```
fgets(str, INFINITY, stdin)
```

Because there is no upper bound to the number of characters read, **gets** may store into memory beyond the bounds of **str**. Therefore, for safety, other functions should be considered instead of **gets**.

EXAMPLE

```
#include <stdio.h>

#define MAXLINE 80
#define SAFETY 40

main()
{
   char line[MAXLINE + SAFETY];

   for (;;){

       /* Instruct user to type up to MAXLINE characters. To  */
       /*  end the program, enter EOF at the input prompt.    */
    printf("Enter a line of data (to %d characters)\n",MAXLINE);

       /* Read data into str from stdin. */
    if (gets(line)){
```

gets Read a String from the Standard Input Stream
(continued)

```
            /* Write string to stdout; check for error. */
         if (puts(line) == EOF) break;
       }
       else break;
      }
   }
```

RELATED FUNCTIONS

`fgets`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

gmtime Break Greenwich Mean Time into Components

POSIX ISO ANSI UNIX

SYNOPSIS

```
#include <time.h>

struct tm *gmtime(const time_t *timep);
```

DESCRIPTION

gmtime converts a **time_t** value to Greenwich Mean Time (GMT), separates it into components and returns a pointer to a **struct tm** containing the results.

RETURN VALUE

gmtime returns a pointer to the broken-down GMT value. The pointer may be to **static** data, which may remain valid only until the next call to **gmtime**, **localtime**, or **ctime**.

CAUTIONS

The pointer returned by **gmtime** may reference **static** storage, which may be overwritten by the next call to **gmtime**, **localtime**, or **ctime**.

gmtime assumes that the value stored in the hardware time-of-day clock is GMT, as specified by 370 standards. If your site uses the time-of-day clock for local time, then **gmtime** returns local time, not Greenwich time, and Greenwich time is unavailable.

DIAGNOSTICS

NULL is returned if GMT is not available or if the argument value is not a valid time.

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main ()
{
   time_t timeval;
   struct tm *now;

   time(&timeval);
   now = gmtime(&timeval);  /* Get current GMT time. */
   if (now->tm_mon == 11 && now->tm_mday == 25)
      puts("Merry Christmas.");
}
```

RELATED FUNCTIONS

localtime

gmtime Break Greenwich Mean Time into Components
(continued)

SEE ALSO

"Timing Functions" on page 2-11

hypot Compute the Hypotenuse function

UNIX

SYNOPSIS

```
#include <lcmath.h>

double hypot(double x,double y);
```

DESCRIPTION

hypot computes the square root of the sum of squares of its two arguments. The value returned by **hypot** is defined by this equation:

$$hypot(x,y) = \sqrt{x^2 + y^2}$$

RETURN VALUE

hypot returns the square root of the sum of the squares of its arguments.

DIAGNOSTICS

The run-time library writes an error message to the standard error file (**stderr**) if the result of the computation would be larger than **HUGE_VAL**. In this case, the function returns **HUGE_VAL**, the largest positive floating-point number that can be represented.

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

PORTABILITY

hypot is portable to many, but not all, non UNIX C implementations.

EXAMPLE

```
#include <stdio.h>
#include <lcmath.h>

#define SVECTOR .7854

main()
{
   double hypov;
   hypov = hypot(SVECTOR,-SVECTOR);
   printf("hypot(%f,%f) = %f \n", SVECTOR,-SVECTOR, hypov);
}
```

RELATED FUNCTIONS

_matherr

SEE ALSO

"Mathematical Functions" on page 2-6

isalnum Alphanumeric Character Test

SYNOPSIS

```
#include <ctype.h>

int isalnum(int c);
```

DESCRIPTION

isalnum tests an integer value to determine whether it is an alphabetic (uppercase or lowercase) or numeric character.

RETURN VALUE

isalnum returns 0 if the character is not alphanumeric, or a nonzero value if it is alphanumeric. If the argument is **EOF**, 0 is returned.

CAUTIONS

The effect of **isalnum** on a noncharacter argument other than **EOF** is undefined. Do not assume that **isalnum** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define MAXLEN 40

main()
{
   char id[MAXLEN+1];
   int i;
   char *text;
   char input[MAXLEN];

   puts("Enter a string of characters (40 at most).");
   text = gets(input);

   puts("Initial alphanumeric characters you entered are:");
   for (i = 0; i < MAXLEN && isalnum(text[i]); i++)  {
      id[i] = text[i];
      putc(id[i]);
   }

   id[i] = '\0';
   putc('\n');
}
```

RELATED FUNCTIONS

isalpha, **isdigit**

isalnum Alphanumeric Character Test
(continued)

SEE ALSO

Chapter 10, ‘‘Localization,’’ in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

isalpha Alphabetic Character Test

ISO ANSI UNIX

SYNOPSIS

```
#include <ctype.h>

int isalpha(int c);
```

DESCRIPTION

isalpha tests an integer value **c** to determine whether it is an alphabetic (uppercase or lowercase) character. In the C locale, **isalpha** returns true only for the characters for which **isupper** or **islower** is true.

RETURN VALUE

isalpha returns 0 if the character is not alphabetic, or a nonzero value if it is alphabetic. If the argument is **EOF**, 0 is returned.

CAUTIONS

The effect of **isalpha** on a noncharacter argument other than **EOF** is undefined. Do not assume that **isalpha** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define MAXLEN 40

main()
{
   char id[MAXLEN+1];
   int i;
   char *text;
   char input[MAXLEN];

   puts("Enter a string (40 characters maximum). ");
   text = gets(input);

   puts("Initial alphabetic characters you entered:");
   for (i = 0; i < MAXLEN && isalpha(text[i]); i++) {
      id[i] = text[i];
      putc(id[i]);
   }

   id[i] = '\0';
   putc('\n');
}
```

RELATED FUNCTIONS

islower, **isupper**

isalpha *(continued)* Alphabetic Character Test

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

isascii ASCII Character Test

SYNOPSIS

```
#include <lctype.h>

int isascii(int c);
```

DESCRIPTION

isascii tests an integer value **c** to determine whether it is the EBCDIC equivalent of a character belonging to the ASCII character set.

RETURN VALUE

isascii returns 0 if the character is not ASCII, or a nonzero value if it is ASCII. If the argument does not have a **char** value, 0 is returned.

CAUTIONS

Do not assume that **isascii** returns either 0 or 1.

PORTABILITY

You should carefully examine the use of **isascii** in a program that you expect to be portable. Traditionally, **isascii** is used to determine whether an integer is a valid character. Unfortunately, in an EBCDIC environment, the name and the function do not mesh well. This implementation defines the **isebcdic** function to test for validity as a **char** value and defines **isascii** as stated above. Therefore, many programs that use **isascii** should be changed to use **isebcdic** when running on the mainframe, unless the intent is to test for membership in the ASCII character set without regard to the native character set of the hardware on which the program is executed.

IMPLEMENTATION

isascii is implemented by a true function. **isascii** tests a character to see whether it is the EBCDIC equivalent of a character in the ASCII character set. This does not produce the same value as the following statement:

```
((unsigned) c) < 128
```

This statement frequently implements **isascii** in a C implementation that uses ASCII as its native character set.

Also note that **isascii('\n')** has the value 0.

isascii ASCII Character Test
(continued)

EXAMPLE

This example tests for printable ASCII characters:

```
#include <lctype.h>
#include <stdio.h>

main()
{
   char input;

   puts("Enter a character: ");
   input = getc();
   if (isascii(input) && !iscntrl(input)){
      puts(" The character you typed is standard ASCII, ");
      puts(" and it is not a control character.");
   }
   else if(!isascii(input) || iscntrl(input){
      puts(" The character you typed is not a standard ASCII");
      puts(" character, or it is a control character.");
   }
}
```

RELATED FUNCTIONS

isebcdic

SEE ALSO

"Character Type Macros and Functions" on page 2-1

isatty Test for Terminal File

POSIX UNIX

SYNOPSIS

```
#include <lcio.h>

int isatty(int fn);
```

The syntax for the POSIX implementation is

```
#include <sys/types.h>
#include <unistd.h>

int isatty(int fn);
```

DESCRIPTION

`isatty` tests whether the file associated with file number **`fn`** is an interactive terminal. **`isatty`** returns a non-zero value if the file number represents a TSO terminal, a CMS terminal, or an OpenEdition terminal. **`isatty`** returns 0 for the DDname SYSTERM when called under MVS batch.

RETURN VALUE

A nonzero value is returned if the file number is associated with an interactive terminal; otherwise, 0 is returned.

EXAMPLE

```
#include <lcio.h>
#include <stdio.h>

double point[40];

main()
{
   FILE *f;
   int index = 0;
   double sum = 0.0;
   double avg ;
   int nopoints;
   int fn =0;

      /* If stdin is the terminal, (fileno(stdin) is always 0). */
   if (isatty(fn))

      /* Tell user to enter data points - max. = 39.            */
   puts("Enter data points (-1 to indicate end of list).");

   for(;;){
         /* Read number; check for end of file.                 */
      if (scanf("%le", &point[index]) <= 0)
         break;
```

isatty Test for Terminal File
(continued)

```
            if (point[index] == -1) break;
            sum += point[index];
            ++index;
         }

         nopoints = index;
         avg = sum / nopoints;
         printf("%d points read.\n", nopoints);
         printf("%f = average.\n", avg);
      }
```

RELATED FUNCTIONS

fattr, **fstat**, **fterm**

SEE ALSO

"I/O Functions" on page 2-13

iscics Return CICS Environment Information

SYNOPSIS

```
#include <lclib.h>

int iscics(void);
```

DESCRIPTION

The **iscics** function returns an indication to a program about whether the program is running in a CICS environment.

RETURN VALUES

The **iscics** function returns 0 if the program is running in a CICS environment; otherwise, it returns a nonzero value.

EXAMPLE

```
#include <lclib.h>

void main()
{
   if (iscics() == 0) puts("Environment is CICS.");
   else puts("Environment is not CICS.");
}
```

RELATED FUNCTIONS

envname

SEE ALSO

"System Interface and Environment Variables" on page 2-17

iscntrl Control Character Test

SYNOPSIS

```
#include <ctype.h>

int iscntrl(int c);
```

DESCRIPTION

iscntrl tests an integer value **c** to determine whether it is a control character.

RETURN VALUE

iscntrl returns 0 if the character is not a control character, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **iscntrl** on a noncharacter argument other than **EOF** is undefined. Do not assume that **iscntrl** returns either 0 or 1.

Note: For some EBCDIC characters, neither **iscntrl(c)** nor **isprint(c)** is true, even though this identity is sometimes used as a definition of **isprint**.

PORTABILITY

You should carefully examine the **iscntrl** function when using it in a program that is expected to be portable. IBM uses the words *control character* to designate characters between **0x00** and **0x3f**, as well as **0xff**. This implementation defines **iscntrl('\xff')** as false.

IMPLEMENTATION

iscntrl is implemented by a macro. **iscntrl** tests a character to see whether it is less than a blank in the EBCDIC collating sequence. This is true for the EBCDIC equivalents of all ASCII control characters.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

main()
{
   char *buf;

   buf = "Hello World. \n This is a test. ";

   do {
   if (!iscntrl(*buf))
      putchar(*buf);
```

iscntrl Control Character Test
(continued)

```
      else
         putchar('*');
      buf++;
      } while (*buf);
      putchar("\n");
   }
```

RELATED FUNCTIONS

`isspace`

SEE ALSO

"Character Type Macros and Functions" on page 2-1

iscsym Test for Valid C Identifier Symbol

SYNOPSIS

```
#include <lctype.h>

int iscsym(int c);
```

DESCRIPTION

iscsym tests an integer value to determine whether it is a character that can appear in a C identifier (after the first character). This implementation includes the uppercase and lowercase alphabetic characters, the digits, and the underscore as valid characters.

RETURN VALUE

iscsym returns 0 if the character is not a valid character in a C identifier, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **iscsym** on a noncharacter argument other than **EOF** is undefined. Do not assume that **iscsym** returns either 0 or 1.

EXAMPLE

See **iscsymf** for an example of this function.

RELATED FUNCTIONS

iscsymf

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

iscsymf Test for Valid C Identifier Initial Symbol

SYNOPSIS

```
#include <lctype.h>

int iscsymf(int c);
```

DESCRIPTION

iscsymf tests an integer value to determine whether it is a character that can appear as the first character of a C identifier. For this implementation, the uppercase and lowercase alphabetic characters and the underscore are included.

RETURN VALUE

iscsymf returns 0 if the character is not a valid first character in a C identifier, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **iscsymf** on a noncharacter argument other than **EOF** is undefined. Do not assume that **iscsymf** returns either 0 or 1.

EXAMPLE

```
#include <lctype.h>
#include <stdio.h>

#define IDMAX 40

main()
{
   char id[IDMAX+1];
   int i;
   char *text;
   char input[IDMAX];

   puts("Enter any identifier (no more than 40 characters long).");
   text = gets(input);

      /* Copy a C identifier from text to id. */
   if (iscsymf(text[0])) {
      id[0] = text[0];

      for (i = 1; i < IDMAX && iscsym(text[i]); ++i)
         id[i] = text[i];

      id[i] = '\0';

      printf("The identifier is copied as %s\n", id);
   }
   else
      puts("The first character of identifier is not acceptable.);
}
```

iscsymf Test for Valid C Identifier Initial Symbol

(continued)

RELATED FUNCTIONS

`iscsym`

SEE ALSO

Chapter 10, "Localization," in *SAS/C Library Reference, Volume 2*

"Character Type Macros and Functions" on page 2-1

isdigit Test for Numeric Character

SYNOPSIS

```
#include <ctype.h>

int isdigit(int c);
```

DESCRIPTION

`isdigit` tests an integer value to determine whether it is a numeric character (digit).

RETURN VALUE

`isdigit` returns 0 if the character is not a digit, or a nonzero value if it is. If the argument is **`EOF`**, 0 is returned.

CAUTION

The effect of **`isdigit`** on a noncharacter argument other than **`EOF`** is undefined. Do not assume that **`isdigit`** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define IDMAX 40

main()
{
   char id[IDMAX+1];
   char *text;
   char input[IDMAX];
   int i;

   puts("Enter a string of numbers/characters (maximum 40):");
   text = gets(input);

      /* Copy a string of digits from text to id. */
   for (i = 0; i < IDMAX && isdigit(text[i]); ++i)
      id[i] = text[i];

   id[i] = '\0';
      /* Only the digits should be copied. */
   printf("You first entered these digits: %s\n", id);
}
```

isdigit Test for Numeric Character
(continued)

RELATED FUNCTIONS

`isxdigit`

SEE ALSO

"Character Type Macros and Functions" on page 2-1

isebcdic EBCDIC Character Test

SYNOPSIS

```
#include <lctype.h>

int isebcdic(int c);
```

DESCRIPTION

isebcdic tests an integer value **c** to determine whether it is a character belonging to the EBCDIC character set.

RETURN VALUE

isebcdic returns 0 if the integer is not an EBCDIC character (that is, if it is not between 0 and 255). If **c** is an EBCDIC character, **isebcdic** returns a nonzero value.

CAUTION

Do not assume that **isebcdic** returns either 0 or 1.

PORTABILITY

isebcdic was devised as a replacement for the **isascii** function in an EBCDIC environment. See **isascii** for further information.

EXAMPLE

This example tests for printable EBCDIC characters:

```
#include <lctype.h>
#include <stdio.h>

void main(void)
   {
   int count;
   int candidate;

   for (count = 0; count < 50; count++) {
      candidate = rand() / 32;
      printf("%d %s to the EBCDIC character set\n", candidate,
         isebcdic(candidate) ? "belongs" : "does not belong");
      }
   }
```

RELATED FUNCTIONS

toebcdic

SEE ALSO

"Character Type Macros and Functions" on page 2-1

isgraph Graphic Character Test

SYNOPSIS

```
#include <ctype.h>

int isgraph(int c);
```

DESCRIPTION

isgraph tests an integer value **c** to determine whether it is a graphic character. (See IMPLEMENTATION below for a discussion of how a graphic character is defined in the 370 environment.)

RETURN VALUE

isgraph returns 0 if the character is not a graphic character, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **isgraph** on a noncharacter argument other than **EOF** is undefined. Do not assume that **isgraph** returns either 0 or 1.

Note: For some EBCDIC characters, neither **iscntrl(c)**, **isspace(c)**, nor **isgraph(c)** is true, even though this identity is sometimes used as a definition of **isgraph**. If **isprint(c)** is true, either **isspace(c)** or **isgraph(c)** is also true.

IMPLEMENTATION

Not all characters considered printable in ASCII are considered printable in EBCDIC.

In the 5370 locale, **isgraph** returns a non-zero value for nonblank characters that are present on the 1403 PN print train. These characters include the digits and letters, plus these special characters:

```
| @ # $ % ¬ * ( ) - _ = + : ; " ' , . / ? < > &
```

This is the set of characters whose printability is guaranteed, regardless of device type. Note that a number of characters used by the C language, including the backslash, the exclamation point, the brackets, and the braces, are not included as graphic characters according to this definition.

In the POSIX locale, **isgraph** returns the results that are expected in an ASCII environment.

isgraph *(continued)*

Graphic Character Test

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>

main()
{
   char *str;
   char input[20];
   size_t len;
   int gcount = 0;

   puts("Enter a string:");
   str = gets(input);

   len = strlen(str);

      /* Test whether all characters in a string are graphic. */
   while (isgraph(*str)) {
      ++gcount;
      ++str;
   }
   if (len == gcount)
      puts("The string entered is entirely graphic.");
   else
      puts("String is not entirely graphic.");
}
```

RELATED FUNCTIONS

isprint, **ispunct**

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

islower Lowercase Alphabetic Character Test

SYNOPSIS

```
#include <ctype.h>

int islower(int c);
```

DESCRIPTION

islower tests an integer value **c** to determine whether it is a lowercase alphabetic character.

RETURN VALUE

islower returns 0 if the character is not a lowercase alphabetic character, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **islower** on a noncharacter argument other than **EOF** is undefined. The definition of a lowercase character is locale dependent. Do not assume that **islower** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

void main()
{
   char id[21];
   char *text;
   int i;

   text = "passwordTESTING";

      /* Copy uppercase "identifier" from text to id. */
   for (i = 0; i < 20 && islower(text[i]); ++i)
      id[i] = text[i];
   id[i] = '\0';

      /* Only the word "password" should be copied. */
   puts( id);
}
```

RELATED FUNCTIONS

isalpha, **isupper**, **tolower**, **toupper**

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

isprint Printing Character Test

SYNOPSIS

```
#include <ctype.h>

int isprint(int c);
```

DESCRIPTION

`isprint` tests an integer value **`c`** to determine whether it is a printing character. (See IMPLEMENTATION below for a discussion of the definition of this concept in the 370 environment.)

RETURN VALUE

`isprint` returns 0 if the character is not a printing character, or a nonzero value if it is. If the argument is **`EOF`**, 0 is returned.

CAUTION

The effect of **`isprint`** on a noncharacter argument other than **`EOF`** is undefined. Do not assume that **`isprint`** returns either 0 or 1.

Note: For some EBCDIC characters, neither **`iscntrl(c)`** nor **`isprint(c)`** is true, even though this identity is sometimes used as a definition of **`isprint`**.

IMPLEMENTATION

Not all characters considered printable in ASCII are considered printable in EBCDIC.

In the 5370 locale, **`isprint`** returns a nonzero value for characters that are present on the 1403 PN print train. These characters include the digits and letters, the blank, and these special characters:

```
| @ # $ % ¬ * ( ) - _ = + : ; " ' , . / ? < > &
```

This is the set of characters whose printability is guaranteed, regardless of device type. Note that a number of characters used by the C language, including the backslash, the exclamation point, the brackets, and the braces, are not included as printing characters according to this definition.

In the POSIX locale, **`isprint`** returns the results that are expected in an ASCII environment.

isprint Printing Character Test
(continued)

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

main()
{
   char *str, *string;
   char input[20];

   puts("Enter a string (max: 20 characters).");
   puts("Note: Do not enter the character '*':");
   string = gets(input);

      /* Test whether all characters in a word are printable. */
   str = string;
   do {
      if (isprint(*str))
         putchar(*str);
      else
         putchar('*');
      ++str;
      } while(*str);
   puts("/nAll unprintable characters have been replaced by '*'.");
}
```

RELATED FUNCTIONS

isgraph, **ispunct**

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

ispunct Punctuation Test

SYNOPSIS

```
#include <ctype.h>

int ispunct(int c);
```

DESCRIPTION

ispunct tests an integer value **c** to determine whether it is punctuation. (See IMPLEMENTATION below for a discussion of the definition of this concept in the 370 environment.)

RETURN VALUE

ispunct returns 0 if the character is not punctuation, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **ispunct** on a noncharacter argument other than **EOF** is undefined. Do not assume that **ispunct** returns either 0 or 1.

Note: For some EBCDIC characters, the return value of **iscntrl(c)**, **isspace(c)**, **isalnum(c)**, or **ispunct(c)** is not true, even though this identity is sometimes used as a definition of **ispunct**. If **isprint(c)** is true, either **isspace(c)**, **isalnum(c)**, or **ispunct(c)** is also true.

IMPLEMENTATION

Not all characters considered printable in ASCII are considered printable in EBCDIC.

In the 5370 locale, **ispunct** returns nonzero for nonblank characters other than the digits and letters that are present on the 1403 PN print train; that is, **ispunct** returns nonzero for these special characters:

```
| @ # $ % ¬ * ( ) - _ = + : ; " ' , . / ? < > &
```

This is the set of characters whose printability is guaranteed, regardless of device type. Note that a number of characters used by the C language, including the backslash, the exclamation point, the brackets, and the braces, are not included as punctuation according to this definition.

In the POSIX locale, **ispunct** returns the results that are expected in an ASCII environment.

ispunct Punctuation Test
(continued)

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

main()
{
   char *str, *string;
   char input[40];

   puts("Enter a string of characters (40 at most), "
        "(preferably, punctuation characters):");
   string = gets(input);

      /* Test whether all characters in string */
      /* are punctuation characters.           */
   str = string;

   do {
      if (ispunct(*str))
         putchar(*str);
      else
         putchar('X');
      ++str;
   } while(*str);
   puts("\n All characters that are not punctuation characters "
        "have been replaced by 'X'.");
}
```

RELATED FUNCTIONS

isgraph, **isprint**

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

isspace White Space Test

SYNOPSIS

```
#include <ctype.h>

int isspace(int c);
```

DESCRIPTION

isspace tests an integer value **c** to determine whether it is white space; that is, a blank, tab, new line, carriage return, form feed, or vertical tab character. In the 5370 and POSIX locales, **isspace** returns true only for standard white-space characters.

RETURN VALUE

isspace returns 0 if the character is not white space, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **isspace** on a noncharacter argument other than **EOF** is undefined. Do not assume that **isspace** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>

main()
{

   int c, n;
   puts("Enter a character when prompted.. even spaces are OK.");
   for (n = 1;; n++ )  {
      puts("Enter now (q to quit):  ");
      c = getchar();
      if (c == 'q') exit(0);
      printf("The character you entered %s a space character\n",
             isspace(c) ? "is" : "is not");
      if (c != '\n')
         getchar();                      /* Read the new line. */
   }
}
```

RELATED FUNCTIONS

iscntrl, **isgraph**

SEE ALSO

"Character Type Macros and Functions" on page 2-1

isupper Uppercase Alphabetic Character Test

SYNOPSIS

```
#include <ctype.h>

int isupper(int c);
```

DESCRIPTION

isupper tests an integer value **c** to determine whether it is an uppercase alphabetic character.

RETURN VALUE

isupper returns 0 if the character is not an uppercase alphabetic character, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **isupper** on a noncharacter argument other than **EOF** is undefined. The definition of an uppercase character is locale dependent. Do not assume that **isupper** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define MAXLEN 40

main()
{
   char id[MAXLEN +1];
   char *text;
   char input[MAXLEN +1];
   int i;

   printf("Enter a string (maximum of %d characters):\n", MAXLEN);
   text = gets(input);

      /* Copy uppercase "identifier" from text to id.           */
   for (i = 0; i < 40 && isupper(text[i]); ++i)
      id[i] = text[i];

   id[i] = '\0';

      /* Only initial uppercase characters get copied.          */
   printf("%s\n", id);
}
```

isupper Uppercase Alphabetic Character Test
(continued)

RELATED FUNCTIONS

`isalpha`, **`islower`**, **`tolower`**, **`toupper`**

SEE ALSO

Chapter 10, "Localization," in *SAS/C Library Reference, Volume 2*

"Character Type Macros and Functions" on page 2-1

isxdigit Hexadecimal Digit Test

SYNOPSIS

```
#include <ctype.h>

int isxdigit(int c);
```

DESCRIPTION

isxdigit tests an integer value **c** to determine whether it is a hexadecimal digit (uppercase or lowercase).

RETURN VALUE

isxdigit returns 0 if the character is not a hexadecimal digit, or a nonzero value if it is. If the argument is **EOF**, 0 is returned.

CAUTION

The effect of **isxdigit** on a noncharacter argument other than **EOF** is undefined. Do not assume that **isxdigit** returns either 0 or 1.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define IDLEN 40

main()
{
   char line[IDLEN];
   char id[IDLEN+1];
   char *text;
   int i;

   puts("Enter a string consisting of hexadecimal characters: ");
   text = gets(line);
      /* Copy from text to id.                                 */
   for (i = 0; i < IDLEN ; ++i) {
      if (isxdigit(text[i]))
         id[i] = text[i];
      else
         id[i] = '*';
   }
   id[i] = '\0';

      /* All nonhex characters have been replaced by '*'.      */
   puts("You have entered the following hexadecimal characters:");
   printf("%s\n", id);
}
```

isxdigit Hexadecimal Digit Test
(continued)

RELATED FUNCTIONS

`isdigit`

SEE ALSO

"Character Type Macros and Functions" on page 2-1

j0 Bessel Function of the First Kind, Order 0

UNIX

SYNOPSIS

```
#include <lcmath.h>

double j0(double x);
```

DESCRIPTION

`j0` computes the Bessel function of the first kind with order 0 of the value `x`.

RETURN VALUE

`j0` returns the Bessel function of the first kind with order 0 of the argument `x`, provided that this value is computable.

DIAGNOSTICS

If the magnitude of `x` is too large (|x| >= 6.7465e9), `j0` returns 0.0. In this case, the message “total loss of significance” is also written to `stderr` (the standard error file).

If an error occurs in `j0`, the `_matherr` routine is called. You supply your own version of `_matherr` to suppress the diagnostic message or modify the value returned.

EXAMPLE

This example computes the Bessel function of the first kind, of order 0 at x = 5 using `j0`:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;
   y = j0(5.);
   printf("j0(5.) = %lf\n", y);
}
```

RELATED FUNCTIONS

`j1`, **`jn`**, **`_matherr`**, **`y0`**, **`y1`**, **`yn`**

SEE ALSO

“Mathematical Functions” on page 2-6

j1 Bessel Function of the First Kind, Order 1

UNIX

SYNOPSIS

```
#include <lcmath.h>

double j1(double x);
```

DESCRIPTION

j1 computes the Bessel function of the first kind with order 1 of the value **x**.

RETURN VALUE

j1 returns the Bessel function of the first kind with order 1 of the argument **x**, provided that this value is computable.

DIAGNOSTICS

If the magnitude of **x** is too large (|x| >= 6.7465e9), **j1** returns 0.0. In this case, the message "total loss of significance" is also written to **stderr** (the standard error file).

If an error occurs in **j1**, the **_matherr** routine is called. You supply your own version of **_matherr** to suppress the diagnostic message or modify the value returned.

EXAMPLE

This example computes the Bessel function of the first kind, of order 1 at x = 5 using **j1**:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;
   y = j1(5.);
   printf("j1(5.) = %lf\n", y);
}
```

RELATED FUNCTIONS

j0, **jn**, **_matherr**, **y0**, **y1**, **yn**

SEE ALSO

"Mathematical Functions" on page 2-6

jn Bessel Function of the First Kind, Order *n*

SYNOPSIS

```
#include <lcmath.h>

double jn(int n, double x);
```

DESCRIPTION

jn computes the Bessel function of the first kind with order **n** of the value **x**. The CPU time required to compute the Bessel function increases with increasing values for **n**. For very large values of **n**, the time can be quite large.

RETURN VALUE

jn returns the Bessel function of the first kind with order **n** of the argument **x**, provided that this value is computable.

DIAGNOSTICS

If the magnitude of **x** is too large (|x| >= 6.7465e9), **jn** returns 0.0. In this case, a message indicating “total loss of significance” is also written to **stderr** (the standard error file).

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

EXAMPLE

This example computes the Bessel function of the first kind, of order 7 at x = 5 using **jn**:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;
    y = jn(7, 5.);
    printf("jn(7, 5.) = %lf\n", y);
}
```

RELATED FUNCTIONS

j0, **j1**, **_matherr**, **y0**, **y1**, **yn**

SEE ALSO

“Mathematical Functions” on page 2-6

kdelete Delete Current Record from Keyed File

SYNOPSIS

```
#include <lcio.h>

int kdelete(const void *key, FILE *f);
```

DESCRIPTION

The **kdelete** function deletes the current record from the keyed stream associated with the **FILE** object addressed by **f**. The **key** argument addresses the value of the **key** field for the record to be deleted. If the key of the current record does not match, the record is not deleted and an error is returned. If the **key** pointer is **NULL**, the current record is deleted, and the key is not checked for validity. After deletion of a record, the file is considered to be positioned to the next record in sequence.

RETURN VALUES

The **kdelete** function returns 0 if no error occurs, or a negative value in case of an error.

CAUTION

Records cannot be deleted from an ESDS, or from a path whose base cluster is an ESDS.

EXAMPLE

For an example using **kdelete**, see “VSAM I/O Example” on page 3-81.

RELATED FUNCTIONS

kinsert, **kretrv**

SEE ALSO

“Keyed access to VSAM files” on page 3-74

“I/O Functions” on page 2-13

kgetpos Return Position Information for VSAM File

SYNOPSIS

```
#include <lcio.h>

int kgetpos(FILE *f, fpos_t *pos);
```

DESCRIPTION

The **kgetpos** function determines the file position of the current record of the keyed stream associated with the **FILE** object addressed by **f** and stores it in the object addressed by **pos**. This object is of type **fpos_t**, which is defined in **<stdio.h>**. If no current record is defined when **kgetpos** is called, **kgetpos** fails.

The **fpos_t** type is composed of two fields, **_recaddr** and **_offset**. The file position returned by **kgetpos** has the control interval number for the current record in **_recaddr**, and the offset of the record in the control interval in **_offset**.

RETURN VALUES

The **kgetpos** function returns 0 if successful, or a negative number if an error occurred.

CAUTION

The position of a record changes due to file updates. Therefore, you should not record file positions for later use, unless the file contents are not subject to change.

RELATED FUNCTIONS

fgetpos

SEE ALSO

"Keyed access to VSAM files" on page 3-74

"I/O Functions" on page 2-13

kill Send Signal to Process

SYNOPSIS

POSIX UNIX

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

DESCRIPTION

kill sends a signal to an OpenEdition process. **pid** is the process ID of the recipient. If **pid** is greater than 0, **kill** sends a signal to a process whose ID equals **pid**. If **pid** is 0, **kill** sends the signal to all processes whose process group ID is equal to that of the sender, for which the sender has the necessary privileges to send a signal. If **pid** is −1, **kill** returns a −1. If **pid** is less than −1, **kill** sends a signal to all processes whose process group ID equals the absolute value of **pid**, for which the sender has the necessary privileges.

sig is the signal. **sig** must be 0, or a signal number defined in **<signal.h>**. The signal number must be one recognized by OpenEdition, not a signal defined by SAS/C. If **sig** is 0, **kill** performs error checking only, and does not send a signal.

A process can also send a **kill** signal to itself. In this case, at least one pending unblocked signal is delivered to the sender if the signal is not blocked or ignored.

You can only use the **kill** function to send a signal supported by OpenEdition; however, the signal need not have been assigned to OpenEdition by **oesigsetup**.

RETURN VALUE

kill returns a 0 if it has permission to send a signal. **kill** returns a −1 if it is not successful.

EXAMPLE

The following example uses **fork** to create a new process, and it uses **kill** to terminate the new process if it fails to terminate in ten seconds. This example uses **oesigsetup** to force the signals SIGALRM and SIGTERM to be managed by OpenEdition:

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <lcsignal.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

static void dospin(void);          /* child process code         */

static void alrmhdlr(int);         /* SIGALRM handler            */
static void termhdlr(int);         /* SIGTERM handler            */
```

kill Send Signal to Process
(continued)

```
static pid_t child;
static volatile int counter;       /* counter for child process  */

main() {
   int status;
   pid_t ended;
   sigset_t oesigs, sascsigs;

   sigemptyset(&oesigs);
   sigemptyset(&sascsigs);
   sigaddset(&oesigs, SIGALRM);
   sigaddset(&oesigs, SIGTERM);
   oesigsetup(&oesigs, &sascsigs); /* Define SIGALRM and SIGTERM */
                                   /*  as OE-managed signals.    */

   child = fork();                 /* Create child process.      */
   if (child == -1) {
      perror("fork error");
      exit(EXIT_FAILURE);
   }
   if (child == 0) dospin();       /* This runs in the child.    */
   else {                         /* This runs in the parent.   */
      signal(SIGALRM, &alrmhdlr);
      alarm(10);                   /* Set alarm for 10 seconds.  */
      ended = wait(&status);
      if (ended == -1 && errno != EINTR) {
                                /* Check for non-signal failure. */
         perror("wait error");
         abort();
      }
      exit(0);
   }
}

void alrmhdlr(int signum) {       /* parent SIGALRM handler     */
   pid_t ended;
   int rc;
   int status;

   rc = kill(child, SIGTERM);      /* Send SIGTERM to child.     */
   if (rc != 0) {
      perror("kill");
      abort();
   }
   ended = wait(&status);          /* Wait for it to quit.       */
   if (ended == -1 && errno != EINTR) {
                                /* Check for non-signal failure. */
      perror("wait error");
      abort();
   }
   return;                         /* to point of signal         */
}
```

kill Send Signal to Process
(continued)

```
void dospin(void) {
   signal(SIGTERM, &termhdlr);     /* Define SIGTERM handler.    */
   for (;;) {          /* busy loop, waiting for SIGTERM signal */
      ++counter;
      if (counter < 0) {   /* Avoid looping absolutely forever. */
         fputs("Counter overflowed, child terminating!\n", stderr);
         exit(EXIT_FAILURE);
      }
      sigchk();               /* Make sure signal is discovered. */
   }
}

void termhdlr(int signum) {  /* handler for termination request */
   fprintf(stderr, "Termination signal received, counter = %d\n",
           counter);
   exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

abort, **raise**, **siggen**

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

kinsert Insert Record into Keyed File

SYNOPSIS

```
#include <lcio.h>

int kinsert(const void *rec, size_t length, void *key, FILE *f);
```

DESCRIPTION

The **kinsert** function adds a new record to the keyed stream associated with the **FILE** object addressed by **f**. The **rec** pointer addresses the record to be written. The **length** argument indicates the length of the record to be written (including the 4-byte key prefix for ESDS or RRDS files). The **key** pointer addresses the key data for the new record. If **key** is a **NULL** pointer, the key is obtained from **rec**.

For an ESDS file, the key for a new record is assigned by VSAM, not by the program. For these files, the contents of the area addressed by **key** are ignored, but the key assigned by VSAM is stored in this area when **kinsert** returns, if **key** is not **NULL**.

After successful completion of **kinsert**, the file is positioned to the record following the one inserted.

RETURN VALUES

The **kinsert** function returns 0 if it is successful, or a negative value if it is unsuccessful.

DIAGNOSTICS

Most VSAM files do not permit more than one record with the same key. If you attempt to add a record with a duplicate key to such a file, **kinsert** returns a negative number, and the file's error flag is set. However, no diagnostic message is issued by the library because this error is frequently expected in working programs. The external variable **errno** is set to **EDUPKEY** for this condition, enabling the program to distinguish this condition from other errors.

If the run-time option **=warning** is in effect, a run-time diagnostic is generated for this condition.

EXAMPLE

For an example using the **kinsert** function, see "VSAM I/O Example" on page 3-81.

RELATED FUNCTIONS

kdelete, **kreplace**

SEE ALSO

"Keyed access to VSAM files" on page 3-74

"I/O Functions" on page 2-13

kreplace Replace Record in Keyed File

SYNOPSIS

```
#include <lcio.h>

int kreplace(const void *rec, size_t length, FILE *f);
```

DESCRIPTION

The **`kreplace`** function replaces the current record of the keyed stream associated with the **`FILE`** object addressed by **`f`** with the data addressed by the **`rec`** argument. The argument **`length`** specifies the length of the replacement record. The new record must not change the key. Additionally, in an ESDS or RRDS, the record length must not be changed.

Replacement of a record does not change the file position. However, the updated record is no longer current and must be retrieved again before another update.

RETURN VALUES

The **`kreplace`** function returns 0 if no error occurs, or a negative value if an error occurs.

EXAMPLE

For an example of using **`kreplace`**, see "VSAM I/O Example" on page 3-81.

RELATED FUNCTIONS

`kdelete`, **`kinsert`**, **`kretrv`**

SEE ALSO

"Keyed access to VSAM files" on page 3-74

"I/O Functions" on page 2-13

kretrv Retrieve Next Record from Keyed File

SYNOPSIS

```
#include <lcio.h>

int kretrv(void *rec, void *key, int flags, FILE *f);
```

DESCRIPTION

The **`kretrv`** function retrieves the next record from the keyed stream associated with the **`FILE`** object addressed by **`f`**. The **`rec`** function addresses the area into which the record is to be read. **`key`** addresses an area in which the key of the record is to be stored. **`key`** may be specified as **`NULL`**, in which case the key is not stored. However, the key can always be retrieved from the record itself.

The **`flags`** argument is an integer that specifies various options through flag bits. Any combination of the following bits can be specified:

`K_backwards`
: indicates that the record before the current record is to be retrieved.

`K_noupdate`
: indicates that the program will not delete or replace this record.

If you do not specify **`K_backwards`**, **`kretrv`** always retrieves the next record. If **`K_noupdate`** is not specified, the record is always retrieved for update (if the file's open mode permits writing).

For a file with duplicate keys, records with the same key are always returned in the order in which they were added to the file, whether or not **`K_backwards`** is specified. **`K_backwards`** only affects the order in which records with different keys are retrieved. See CAUTION for additional restrictions relating to files with duplicate keys.

RETURN VALUES

If successful, **`kretrv`** returns the length of the record retrieved, including 4 bytes for the key prefix for an ESDS or RRDS data set. If **`kretrv`** fails to return a record because the file is positioned to end of file (or beginning of file if you specify **`K_backwards`**), a return code of 0 is returned. If no record is retrieved due to an error, a negative number is returned.

CAUTION

The first **`kretrv`** call after a call to **`ksearch`** should specify the same flag settings if possible. Specifying different options is not an error, but will necessitate substantial additional processing.

When processing a file with duplicate keys, you are not permitted to switch between forward and backward retrieval except with a call to **`ksearch`**.

When the same VSAM cluster is accessed by several streams, with at least one of the streams permitting writing, retrieving a record that the program has previously updated with **`K_noupdate`** set may retrieve an obsolete copy of the record. In applications where the same file is opened several times and obtaining the most recent copy of each record is important, you should not set **`K_noupdate`**. This is true even if you do not intend to modify the record. Alternately, if your program can detect this out-of-sync condition, you can use **`afflush`** to write all buffers to disk, thus ensuring that the record on disk and any copies of the record in memory are identical.

kretrv Retrieve Next Record from Keyed File

(continued)

EXAMPLE

See “VSAM I/O Example” on page 3-81.

RELATED FUNCTIONS

`ksearch`, `kseek`

SEE ALSO

“Keyed access to VSAM files” on page 3-74

“I/O Functions” on page 2-13

ksearch Search Keyed File for Matching Record

SYNOPSIS

```
#include <lcio.h>

int ksearch(const void *key, size_t keylen, int flags, FILE *f);
```

DESCRIPTION

The **ksearch** function searches the keyed stream associated with the **FILE** object addressed by **f** for a record matching the **key** specification. The key can either be a full key or a generic key. If the key is generic, the key length is specified by the **keylen** argument, and only the first **keylen** characters of each key are considered during the search. If the key is not generic, you should specify **keylen** as 0, and all characters of the key are considered. Generic key searches are supported only for KSDS data sets.

The **flags** argument is an integer that specifies various options through flag bits. Any combination of the following bits can be specified:

K_exact
: specifies that a matching record must match the requested key exactly (up to **keylen** characters for a generic key search). If **K_exact** is not specified, the first record found with a key greater than or equal to the key specified (less than or equal to a backward search) is considered to match. **K_exact** must be specified for ESDS searches.

K_backwards
: indicates that the search is to be performed in descending key order rather than ascending key order.

K_noupdate
: indicates that the program does not attempt to replace or delete records located by the search. You can override this option by a later call to **kretrv**.

For files with duplicate keys, you must specify **K_exact** if you specify **K_backwards**, and you must not use a generic key.

The **ksearch** function positions the file to the first record with matching keys, but does not copy the record. You must call **kretrv** to access the record. If **ksearch** fails to locate a matching record, the file is positioned to the point at which a matching record would be inserted. For example,

- A full key-forward search without **K_exact** sets positions to the end of file.
- A full key-backward search without **K_exact** sets positions to the start of file.
- A full-key search with **K_exact** sets positions to the first record after the specified key.
- A generic-key search with **K_exact** sets positions to the first record whose partial key follows the generic key.

ksearch Search Keyed File for Matching Record
(continued)

RETURN VALUES

The **ksearch** function returns the number of records found by the search. This number is 1, unless the file permits duplicate keys, in which case the number of records with the key of the first record found is returned. The **ksearch** function returns 0 if no matching record is found, or a negative value if an error occurred.

CAUTION

Keys are always compared as character arrays, even if they are declared as some other type. For instance, if the **key** field of a VSAM file has type **int**, and an inexact search is made for a key greater than or equal to 4096, a record with a key of −1 may be returned because the search key, \x00\x00\x10\x00 (in hexadecimal notation) as a character array, is less than the record key of \xff\xff\xff\xff.

EXAMPLE

This example locates a record in a VSAM file with a specific key. If the record is not found, it determines the next and previous records and prints their keys. The example function returns 0 if no record is found, or a negative value if an error occurs:

```
#include <lcio.h>

extern FILE *vsam;

int findrec(void *key, void *rec)
{
   int rc;
   char nearkey [20] ;

   rc = ksearch(key, 0, K_exact | K_noupdate, vsam);
   if (rc < 0) return rc;         /* If error, return.        */
   if (rc > 0) {
      rc = kretrv(rec, NULL, K_noupdate, vsam);
      printf("Record %.20s found.\n", key);
      return rc;                  /* Return length of record. */
   }
   printf("Record %.20s not found.\n", key);
       /* Search for greater key.                             */
   rc = ksearch(key, 0, K_noupdate, vsam);

   if (rc < 0) return rc;
   if (rc == 0)                   /* no greater record        */
      puts("No record following this key.");
   else {
      rc = kretrv(rec, nearkey, K_noupdate, vsam);
      if (rc < 0) return rc;
      printf("Following record key: %.20s\n", nearkey);
   }
```

ksearch Search Keyed File for Matching Record
(continued)

```
      /* Retrieve previous record.                            */
   rc = kretrv(rec, nearkey, K_backwards | K_noupdate, vsam);
   if (rc < 0) return rc;
   if (rc == 0)                  /* no previous record       */
      puts("No record preceding this key.");
   else {
      rc = kretrv(rec, nearkey, K_backwards | K_noupdate, vsam);
      if (rc < 0) return rc;
      printf("Preceding record key: %.20s\n", nearkey);
   }
   return 0;                     /* Show record not found.   */
}
```

RELATED FUNCTIONS

kretrv, **kseek**

SEE ALSO

"Keyed access to VSAM files" on page 3-74

"I/O Functions" on page 2-13

kseek Reposition a Keyed Stream

SYNOPSIS

```
#include <lcio.h>

int kseek(FILE *f, int pos);
```

DESCRIPTION

The **kseek** function repositions the stream associated with the **FILE** object addressed by **f**, as specified by the value of **pos**, and releases control of any current record. The **pos** value must be either **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**. (These constants are defined in **<stdio.h>**.) Each of these values refers to a specific location in the file, as follows:

SEEK_SET	refers to the start of file.
SEEK_CUR	refers to the current file position.
SEEK_END	refers to the end of file.

If the current record is read for update, the record is released and is read for update by another **FILE** object. Even though **kseek** with **pos** value **SEEK_CUR** causes no change in the file's positioning, it is still useful to release control of the current record.

RETURN VALUES

The **kseek** function returns 0 if successful, or a negative value if an error occurred.

EXAMPLE

This example retrieves the last record in a VSAM file:

```
#include <lcio.h>

int getlast(FILE *f, void *buf)
{
   int length;
   rc = kseek(f, SEEK_END);
   if (rc < 0) return rc;    /* in case it fails */
      /* Retrieve backwards from EOF.           */
   length = kretrv(buf, NULL, K_BACKWARDS, f);
   return length;
}
```

RELATED FUNCTIONS

ksearch

SEE ALSO

"Keyed access to VSAM files" on page 3-74

"I/O Functions" on page 2-13

ktell Return RBA of Current Record

SYNOPSIS

```
#include <lcio.h>

unsigned long ktell(FILE *f);
```

DESCRIPTION

The **`ktell`** function returns the RBA of the current record of the stream associated with the **`FILE`** object addressed by **`f`**. If there is no current record when **`ktell`** is called, an error indication is returned.

For an ESDS, a record's RBA is stored in its key field, so in general, using **`ktell`** is not necessary with an ESDS.

RETURN VALUES

The **`ktell`** function returns the RBA of the current record if it is successful, or **`-1UL`** if an error occurs.

RELATED FUNCTIONS

`kgetpos`, **`kseek`**

SEE ALSO

"Keyed access to VSAM files" on page 3-74

"I/O Functions" on page 2-13

labs Integer Conversion: Absolute Value

SYNOPSIS

```
#include <stdlib.h>

long int labs(long int j);
```

DESCRIPTION

labs returns the absolute value of a **long int**.

RETURN VALUE

labs returns the absolute value of its argument. Both the result and the argument are of **long int** type.

IMPLEMENTATION

labs is implemented as a macro that invokes the built-in **abs** function.

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   long int a, c;  /* The variable, a, can have a negative value. */
   int b;

   puts("Enter values for a (can be negative) and b:");
   scanf("%ld %d", &a, &b);

   c = labs(a*b);  /* Calculate absolute value.                  */

   printf("The absolute value of their product = %ld\n", c );
}
```

RELATED FUNCTIONS

abs, **fabs**

SEE ALSO

“Mathematical Functions” on page 2-6

ldexp Floating-Point Conversion: Load Exponent

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double ldexp (double y, int i);
```

DESCRIPTION

`ldexp` calculates $y * 2^i$.

RETURN VALUE

`ldexp` returns the value of type **`double`** that results from the calculation.

DIAGNOSTICS

If the return value cannot be represented, a warning message is sent to the standard error file (**`stderr`**). ± **`HUGE_VAL`** is returned if an overflow occurs; 0 is returned if an underflow occurs.

USAGE NOTES

When used with **`frexp`**, the **`ldexp`** function is useful in situations that require repeated multiplication by 2. If the next multiplication causes an overflow or underflow, use **`frexp`** to separate the mantissa from the exponent. This gives you complete control over the exponent and the mantissa, so you can operate on them separately without any loss of precision. When you are finished, use **`ldexp`** to combine the mantissa and exponent again.

RELATED FUNCTIONS

`frexp`, **`_ldexp`**

SEE ALSO

"Mathematical Functions" on page 2-6

_ldexp Fast Implementation of ldexp

SYNOPSIS

```
#include <lcmath.h>

double _ldexp (double y, int i);
```

DESCRIPTION

`_ldexp` calculates $y * 2^i$. It is almost identical to **`ldexp`** but it is faster because no error checking is performed.

RETURN VALUE

`_ldexp` returns the value of type **`double`** that results from the calculation.

DIAGNOSTICS

`_ldexp` generates the SIGFPOFL or SIGFPUFL signal if an error occurs.

PORTABILITY

`_ldexp` is not portable.

IMPLEMENTATION

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

The difference between **`ldexp`** and **`_ldexp`** is that **`_ldexp`** does not check for erroneous arguments, so an ABEND OCC or OCD occurs if the arguments are invalid. Because **`_ldexp`** avoids error checking, it is significantly faster than **`ldexp`**.

RELATED FUNCTIONS

`ldexp`

SEE ALSO

"Mathematical Functions" on page 2-6

ldiv Integer Conversion: Division

SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv(long int numer, long int denom);
```

DESCRIPTION

ldiv computes the quotient and remainder of **numer** divided by **denom**.

RETURN VALUE

div returns a structure of type **ldiv_t**, which contains both the quotient and remainder. The definition of the **ldiv_t** type is

```
typedef struct {
   long rem;
   long quot;
}ldiv_t;
```

The return value is such that

```
numer == quot * denom + rem
```

The sign of **rem** is the same as the sign of **numer**.

EXAMPLE

This example converts an angle in radians to degrees, minutes, and seconds using **ldiv**:

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

main()
{
   double rad, angle;
   long deg, min, sec;
   ldiv_t d;

   puts(" Enter any angle in radians: ");
   scanf("%f", &rad);

      /* Convert angles to seconds and discard fraction. */
   angle = rad * (180 * 60 * 60)/M_PI;

   sec = angle;
   d   = ldiv(sec, 60L);
   sec = d.rem;
   d   = ldiv(d.quot, 60L);
```

ldiv Integer Conversion: Division
(continued)

```
      min = d.rem;
      deg = d.quot;

      printf("%f radians = %ld degrees, %ld, %ld\n", rad,
              deg, min, sec);
   }
```

RELATED FUNCTIONS

`div`

SEE ALSO

"Mathematical Functions" on page 2-6

link Create Link to File

SYNOPSIS

```
#include <unistd.h>

int link(char *oldfile, char *newname);
```

DESCRIPTION

`link` creates a hard link to an existing OpenEdition file. **`oldfile`** is the existing file. If **`oldfile`** is a symbolic link, the link refers to the file that is referenced by the pathname in the symbolic link. **`newname`** is the new pathname. If the old name is removed, the file continues to exist with the new name. You cannot create a link to a directory.

Both **`oldfile`** and **`newname`** must specify OpenEdition HFS filenames. For programs not compiled with the **`posix`** option, style prefixes may be required. See "File Naming Conventions" on page 3-56 for information on specifying OpenEdition filenames.

RETURN VALUE

`link` returns a 0 if it is successful; **`link`** increments the link count that refers to the number of links to the file. **`link`** returns a −1 if it is not successful; the link count is not incremented.

EXAMPLE

This example uses **`link`** and **`unlink`** to change the name of an HFS file. It differs from the **`rename`** function in two ways:

- □ If the new name already exists, it fails and leaves the name unchanged.
- □ It is not atomic, so errors could occur if another program simultaneously modifies the same directory entries.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int chname(char *oldname, char *newname)
{
   int rc;
   rc = link(oldname, newname);
   if (rc != 0) {
      perror("link error");
      return -1;      /* Go no further if the link fails. */
   }
      /* The link worked. Now we can unlink the old name. */
   rc = unlink(oldname);
   if (rc != 0) {
      perror("unlink error");
      return -1;
   }
   return 0;
}
```

link Create Link to File
(continued)

RELATED FUNCTIONS

`symlink`, **`unlink`**

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*
"File Management Functions" on page 2-16

localtime Break Local Time Value into Components

POSIX ISO ANSI UNIX

SYNOPSIS

```
#include <time.h>

struct tm *localtime(const time_t *timep);
```

DESCRIPTION

`localtime` converts a **`time_t`** value to local time, separates it into components, and returns a pointer to a **`struct tm`** containing the results.

Under POSIX, **`localtime`** is affected by time-zone information contained in the TZ environment variable, if it is defined.

RETURN VALUE

`localtime` returns a pointer to the broken-down, local-time value. The pointer may be to **`static`** data, which may remain valid only until the next call to **`gmtime`**, **`localtime`**, or **`ctime`**.

CAUTION

The pointer returned by **`localtime`** may reference **`static`** storage, which may be overwritten by the next call to **`gmtime`**, **`localtime`**, or **`ctime`**.

DIAGNOSTICS

`NULL` is returned if local time is not available or if the argument value is not a valid time.

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main()
{
   time_t timeval;
   struct tm *now;

   time(&timeval);
   now = localtime(&timeval);    /* Get current local time. */
   if (now->tm_mon == 0 && now->tm_mday == 1)
      puts("Happy New Year.");
}
```

RELATED FUNCTIONS

`gmtime`, **`tzset`**

SEE ALSO

"Timing Functions" on page 2-11

log Compute the Natural Logarithm

SYNOPSIS

```
#include <math.h>

double log(double x);
```

DESCRIPTION

log computes the natural log of its argument **x**. The **x** argument must be a positive double-precision, floating-point number. The natural log is the inverse of the exponential function.

RETURN VALUE

log returns the natural log of its argument **x**, expressed as a double-precision, floating-point number.

DIAGNOSTICS

The run-time library writes an error message to the standard error file (**stderr**) if **x** is a negative number or 0. In this case, the function returns **-HUGE_VAL**, the largest negative floating-point number that can be represented.

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

EXAMPLE

This example computes the natural log of 10, using **log**:

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

main()
{
   double y, val;
   val = 10.0;
   y = log(val);
   printf("log(%f) = %f\n", val, y);
}
```

RELATED FUNCTIONS

log10, **_matherr**

SEE ALSO

"Mathematical Functions" on page 2-6

log10 Compute the Common Logarithm

SYNOPSIS

```
#include <math.h>

double log10(double x);
```

DESCRIPTION

`log10` computes the common (base 10) log of its argument **`x`**. The **`x`** argument must be a positive double-precision, floating-point number.

RETURN VALUE

`log10` returns the common log of its argument, expressed as a double-precision, floating-point number.

DIAGNOSTICS

If **`x`** is negative or 0, **`log10`** returns **`-HUGE_VAL`**. In this case, the run-time library also writes an error message to the standard error file (**`stderr`**).

If an error occurs in **`log10`**, the **`_matherr`** routine is called. You supply your own version of **`_matherr`** to suppress the diagnostic message or modify the value returned.

EXAMPLE

This example calculates the common log of RATE, and rounds up using **`log10`**:

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

#define RATE .017

main()
{
   double y;
   y = ceil(log10(RATE));

      /* Print the "order of magnitude" of RATE. */
   printf("ceil(log10(%f)) = %f\n", RATE, y);
}
```

RELATED FUNCTIONS

`log`, **`_matherr`**

SEE ALSO

“Mathematical Functions” on page 2-6

longjmp Perform Nonlocal goto

ISO ANSI UNIX

SYNOPSIS

```
#include <setjmp.h>

void longjmp(jmp_buf env, int code);
```

DESCRIPTION

longjmp returns control to an ancestor routine, passing the value of the integer **code**. The point of return is determined by the contents of **env**, which should be initialized by a call to **setjmp** in the target routine. Note that if the value of **code** is 0, the value returned to the target is 1.

RETURN VALUE

Control does not return from **longjmp**.

CAUTION

longjmp does not change the program's signal mask. Use **sigsetjmp** to save the signal mask, and **siglongjmp** to restore the signal mask to the mask in effect when **sigsetjmp** was called.

ERRORS

If the target is not valid (that is, if the routine that called **setjmp** has terminated), a user ABEND 1204 is issued. If an invalid **env** is not detected, serious (and unpredictable) problems occur.

If you attempt to terminate a routine in a language other than C, a user ABEND 1224 is issued. See Appendix 5, "Using the INDep Option for Interlanguage Communication," in the *SAS/C Compiler and Library User's Guide* for more information.

EXAMPLE

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf env;

main()
{
   int ret;

   if ((ret = setjmp(env)) != 0) {
      fprintf(stderr, "longjmp called with value %d\n", ret);
      exit(1);
   }
   dummy();
   fprintf(stderr, "longjmp was not called.\n");
}

void dummy(void)
```

longjmp Perform Nonlocal goto
(continued)

```
   {
      puts("Entering dummy routine.");
      longjmp(env, 3);
      puts("Never reached.");
   }
```

RELATED FUNCTIONS

`blkjmp`, **`setjmp`**, **`siglongjmp`**

SEE ALSO

"Program Control Functions" on page 2-9

lseek Position a UNIX Style File

POSIX UNIX

SYNOPSIS

```
#include <fcntl.h>

long lseek(int fn, long offset, int pos);
```

The synopsis for the POSIX implementation is

```
#include <unistd.h>

off_t lseek(int fn, off_t offset, int pos);
```

DESCRIPTION

lseek replaces the current offset with a new position in the file. **fn** is the file descriptor. **offset** is the specified byte offset; it can be positive or negative. **pos** is the position from which **offset** is specified. The next I/O operation on the file begins at **offset**.

pos can be one of the following symbols defined in **<unistd.h>** and **<fcntl.h>**:

SEEK_SET	is the beginning of the file; the value is 0.
SEEK_CUR	is the current file offset; the value is 1.
SEEK_END	is the end of the file; the value is 2.

lseek(fn, 0L, SEEK_CUR) determines the current file position without changing it. You can use **lseek** to set the position past the current end of file, but after such positioning, attempts to read data shows end of file. If you call **write** while the file is so positioned, 0s are written into all character positions between the previous end of file and the current position, after which the specified data are written.

If **fn** represents a non-HFS file opened as text, **offset** does not represent a physical offset in bytes. In this case, the **lseek offset** is interpreted according to the same rules as an **fseek offset**. For more information, see "File positioning with standard I/O (fseek and ftell)" on page 3-49.

RETURN VALUE

If successful, **lseek** returns the new file position; otherwise, it returns −1L.

EXAMPLE

This example determines the size of a file by using **lseek**:

```
#include <fcntl.h>

main()
{
   long size;
   int fileno;

   fileno = open("tso:TEST", O_RDONLY);
```

lseek Position a UNIX Style File
(continued)

```
         /* accesses the file as binary */

      if ((size = lseek(fileno, 0L, SEEK_END)) == -1)
         perror("lseek() error");
      else {
         printf("Size of file is: %ld\n", size);
         close(fileno);
      }
   }
```

RELATED FUNCTIONS

`fgetpos`, **`fseek`**, **`fsetpos`**, **`ftell`**, **`ksearch`**, **`kseek`**

SEE ALSO

"File positioning with UNIX style I/O" on page 3-47
"I/O Functions" on page 2-13

_lseek Position an HFS File

DESCRIPTION

`_lseek` is a version of `lseek` designed to operate only on HFS files. `_lseek` runs faster and calls fewer other library routines than `lseek`. See `lseek` for a full description.

`_lseek` is used exactly like the standard `lseek` function. The first argument to `_lseek` must be the file descriptor for an open HFS file.

lstat Get File or Link Status

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int lstat(const char *pathname, struct stat *buf);
```

DESCRIPTION

lstat gets status information about a file or symbolic link. **pathname** is the file. **buf** is the memory area for storing the status information.

The status information is returned in the **stat** structure defined in **<sys/stat.h>**:

mode_t st_mode	indicates the permissions set for the file.
ino_t st_ino	is the file serial number.
dev_t st_dev	is the numeric ID of the device.
nlink_t st_nlink	is the number of links to the file.
uid_t st_uid	is the numeric user ID of the owner of the file.
gid_t st_gid	is the numeric group ID of the file.
off_t st_size	indicates the file size in bytes for regular files.
time_t st_atime	indicates the most recent access time.
time_t st_ctime	indicates the most recent time the file status was changed.
time_t st_mtime	indicates the most recent time the file contents were changed.

The pathname must be specified as an OpenEdition HFS file. For programs not compiled with the **posix** option, a style prefix may be required. See "Low-level and Standard I/O" on page 3-54 for information on specifying OpenEdition filenames.

The **<sys/stat.h>** header file contains a collection of macros that you can use to examine properties of a **mode_t** value from the **st_mode** field. For a list of these macros, see the documentation on the **fstat** function.

RETURN VALUE

lstat returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <time.h>
```

lstat Get File or Link Status

(continued)

```
main()
{

   char flname[] ="link.file";
   char lkname[] ="link.example";
   int fd;
   struct stat info;

      /* Create a file for linking.                          */
   if ((fd = open(flname,O_CREAT)) < 0)
      perror("open() error");
   else {
      close(fd);
         /* Check the status of the file.                    */
      puts("Status before link:");
      system("ls -il link.*");
         /* Create an alternative path for the existing file. */
      if (link(flname,lkname) != 0) {
         perror("link() error");
         unlink(flname);
      }
      else {
         puts("Status after linking:");
         system("ls -il link.*");
         if (lstat(lkname, &info) != 0)
            perror("lstat() error");
         else {
            printf("\nlstat() for link %s returned:\n", lkname);
            printf("inode: %d\n", (int) info.st_ino);
            printf("dev id: %d\n", (int) info.st_dev);
            printf("user id: %d\n", (int) info.st_uid);
            printf("links: %d\n",  info.st_nlink);
            printf("file size: %d\n", (int) info.st_size);
            printf("mode: %08x\n",  info.st_mode);
            printf("created: %s", ctime(&info.st_createtime));
         }
      }
      unlink(flname);
      unlink(lkname);
   }
}
```

RELATED FUNCTIONS

`fstat`, **`stat`**, **`symlink`**

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

“File Management Functions” on page 2-16

malloc Allocate Memory

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);
```

DESCRIPTION

`malloc` allocates a block of dynamic memory of the size requested by **`size`**.

RETURN VALUE

`malloc` 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.

CAUTION

The contents of a memory block on allocation are random.

DIAGNOSTICS

If adequate memory is not available, or if 0 bytes are requested, **`NULL`** is returned.

IMPLEMENTATION

Whenever a block is allocated, the amount actually consumed is 16 bytes more than the amount requested (after rounding up to an integral number of doublewords) due to memory management overhead. If more than 288 bytes are requested (after addition of the overhead), the amount is rounded up to an even multiple of 64 bytes.

Storage is always obtained from the operating system in page multiples and only when necessary. Small blocks are kept separate from large blocks to reduce fragmentation.

Under an XA or ESA operating system, memory allocated by **`malloc`** can reside above the 16-megabyte line for programs that run in 31-bit addressing mode.

Allocation of a large number of small blocks or of blocks that are slightly larger than half a page may result in significant waste of memory. Use of the **`pool`**, **`palloc`**, and **`pfree`** routines is recommended in such cases.

malloc Allocate Memory
(continued)

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *source, *copy;

main()
{
   source = "A simple line for the malloc example ";

      /* Allocate space for a copy for source. */
   copy = malloc(strlen(source) + 1);

      /* Copy if there is space. */
   if (copy){
      strcpy(copy,source);
      puts(copy);
   }
   else puts("malloc failed to allocate memory for copy".);
}
```

RELATED FUNCTIONS

pool, **realloc**, **sbrk**

SEE ALSO

"Memory Allocation Functions" on page 2-10

_matherr Handle Math Function Error

SYNOPSIS

```
#include <lcmath.h>

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

DESCRIPTION

_matherr is called whenever one of the other math functions detects an error. Upon entry, it receives an exception block that describes the error in detail. This structure is defined in **<lcmath.h>**:

```
struct exception {
   int type;       /* error type                     */
   char *name;     /* name of function having error */
   double arg1;    /* first argument                 */
   double arg2;    /* second argument                */
   double retval;  /* proposed return value          */
};
```

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 **_matherr** returns 0, a diagnostic message is written to the standard error file (**stderr**). If **_matherr** returns a nonzero value, the diagnostic message is suppressed, and the calling function is forced to accept a new value from **retval**.

PORTABILITY

Traditional UNIX C compilers support the functionality of **_matherr** 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 _matherr
```

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

_matherr Handle Math Function Error
(continued)

IMPLEMENTATION

The standard version of **_matherr** supplied in the library places the appropriate error number into the external integer **errno** and returns 0. When **_matherr** is called, the function that detected the error places its proposed return value into the exception structure. The 0 return code indicates that the return value should be used.

Supply your own version of **_matherr** 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 **_matherr**, 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.

EXAMPLE

```
#include <lcmath.h>
#include <lcio.h>
#include <lclib.h>

   /* user status flags */
#define ERROR_OK           9000
#define ERROR_WARNING      9001
#define ERROR_SEVERE       9002

   /* global status flag */
int status;

   /* user-defined math status handler */
int _matherr(struct exception *err)
{
   err->retval = -1;

      /* Check to see if an error occurred */
      /* in the call to sqrt.             */
   if (strcmp(err->name, "sqrt") == 0)
      status = ERROR_SEVERE;

      /* Check to see if an error occurred */
      /* in the call to log or log10.      */
   else if (strncmp(err->name, "log", 3) == 0)
      status = ERROR_WARNING;
   else status = ERROR_OK;

   return(1);
}

main()
```

_matherr Handle Math Function Error
(continued)

```
{
   double x, y;

   while (feof(stdin) == 0) {

         /* Read data and echo it. */
      scanf("%f", &x);
      printf("\necho: x = %f\n", x);
      y = cosh(x);

         /* If no unexpected error occurred, print result. */
      if (warning() == 0)
         printf("result = %f\n", y);
      y = log10(x);

         /* If no unexpected error occurred, print result. */
      if (warning() == 0)
         printf("result = %f\n", y);
      y = sqrt(x);

         /* If no unexpected error occurred, print result. */
      if (warning() == 0)
         printf("result = %f\n", y);
   }  /* End while loop. */

   return(0);
}

int warning(void)
{
   if (status == ERROR_SEVERE) {
      printf("A severe error occurred. status condition = %d"
             " ***All processing is terminated***\n", status);
      exit(EXIT_FAILURE);
   }
   else if (status == ERROR_WARNING) {
      puts("An error occurred; processing will continue.");
      status = ERROR_OK;
      return(1);
   }

   status = ERROR_OK;
   return(0);
}
```

_matherr Handle Math Function Error
(continued)

RELATED FUNCTIONS

`quiet`

SEE ALSO

"Mathematical Functions" on page 2-6

max Find the Maximum of Two Integers

SYNOPSIS

```
#include <lclib.h>

int max(int s, int r);
```

DESCRIPTION

max finds the maximum of two integer values, **s** and **r**.

RETURN VALUE

max returns an integer value that represents the maximum of the two arguments.

IMPLEMENTATION

max is a built-in function.

EXAMPLE

```
#include <lcmath.h>
#include <stdio.h>

main()
{
   int num1, num2;     /* numbers to be compared            */
   int result;         /* holds the larger of num1 and num2 */

   puts("Enter num1 & num2 : ");
   scanf("%d %d", &num1, &num2);
   result = max(num1, num2);
   printf("The larger number is %d\n", result);
}
```

RELATED FUNCTIONS

fmax, **fmin**, **min**

SEE ALSO

"Mathematical Functions" on page 2-6

memchr Locate First Occurrence of a Character

SYNOPSIS

```
#include <string.h>

void *memchr(const void *ptr, int ch, size_t n);
```

DESCRIPTION

memchr searches **n** bytes, beginning at the location pointed to by **ptr**, for the first occurrence of **ch**.

RETURN VALUE

The return value is a pointer to the first occurence of **ch**, or **NULL** if the character cannot be found.

CAUTION

The third argument to **memchr** is **size_t**.

See the **memscntb** function description for information on possible interactions between the **memchr**, **memscntb**, or **strscntb** functions.

EXAMPLE

This example counts the number of zero bytes in a 256-byte memory area:

```
#include <string.h>
#include <stdio.h>
#define SIZE 256

main()
{
   char area[SIZE];
   int offset = 0;
   int count = 0;
   char *next, *prev;
   int i;

      /* Every alternate element is assigned zero value. */
   for (i = 0; i <= 255; i+=2) {
      area[i] = '\1';
      area[i+1] = '\0';
   }
   prev = area;
   for (;;) {
      next = memchr(prev, '\0' , 256 - offset);
      if (!next) break;
      ++count;
      prev = next+1;
      offset = (prev - area + 1);
   }
   printf("%d zero bytes found. \n", count);
}
```

memchr Locate First Occurrence of a Character
(continued)

RELATED FUNCTIONS

`strchr`, **`memscan`**

SEE ALSO

“String Utility Functions” on page 2-3

memcmp Compare Two Blocks of Memory

SYNOPSIS

```
#include <string.h>

int memcmp(const void *ptr1, const void *ptr2, size_t n);
```

DESCRIPTION

memcmp compares 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, which is performed using the standard EBCDIC collating sequence.

RETURN VALUE

memcmp 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

The compiler generates inline code for **memcmp** unless **memcmp** is undefined (by an **#undef** statement) to prevent this. The inline code may still call a library routine in special cases (for example, if the length is a variable whose value is larger than 16 megabytes). For more information on optimizing your use of **memcmp**, see "Optimizing Your Use of memcmp, memcpy, and memset" on page 2-5.

Usually, the code generated by **memcmp** uses the CLCL instruction to perform the comparison. If more than 16 megabytes of data are to be compared, the library routine is called, which processes 16M-1 bytes at a time.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

main()
{
   struct large {
      int month;
      int day;
      int year;
   };

   struct large day1, day2, *first, *second;

   day1.month = 7;
   day1.day = 29;
   day1.year = 1993;
   day2.month = 7;
   day2.day = 30;
   day2.year = 1993;
```

memcmp Compare Two Blocks of Memory
(continued)

```
         first = &day1;
         second = &day2;

            /* Compare one structure to another. Note: if structures */
            /*  contain padding between elements, the results may be */
            /*  misleading.                                          */
         if (memcmp((char *)first,(char *)second, sizeof(struct large)))
             puts("structures not equal");
         else
             puts("structures are equal");
      }
```

RELATED FUNCTIONS

memcmpp, **strcmp**, **strncmp**

SEE ALSO

"String Utility Functions" on page 2-3

memcmpp Compare Two Blocks of Memory with Padding

SYNOPSIS

```
#include <lcstring.h>

int memcmpp (const void *arg1, const void *arg2,
             size_t size1, size_t size2, int pad);
```

DESCRIPTION

memcmpp compares two arrays (**arg1** and **arg2**), each of which has an associated size (**size1** and **size2**). The shorter array is treated as if it has the same size as the larger, and all additional bytes have the value specified by **pad**.

RETURN VALUE

memcmpp returns 0 if the two strings are equal, an integer less than 0 if the first string is less than the second, or an integer greater than 0 if the first string is greater than the second.

IMPLEMENTATION

The compiler generates inline code for **memcmpp** unless **memcmpp** is undefined (by an **#undef** statement) to prevent this. The inline code may still call a library routine in special cases (for example, if the length is a variable whose value is larger than 16 megabytes).

Usually, the code generated by **memcmpp** uses the CLCL instruction to perform the comparison. If more than 16 megabytes of data are to be compared, the library routine is called, which processes 16M-1 bytes at a time.

EXAMPLE

```
#include <lcstring.h>
#include <stdlib.h>
#include <stdio.h>

#define MAXLEN 100

main()
{
   char line[MAXLEN];
   char *cmdname;
   int cmdlen;

   command:
   puts("enter command:");
   cmdname = gets(line);
   cmdlen = strlen(cmdname);
```

memcmpp Compare Two Blocks of Memory with Padding
(continued)

```
         /* If the value in cmdname is "exit", possibly       */
         /* followed by trailing blanks, terminate execution. */
      if (memcmpp(cmdname, "exit", cmdlen, 4, ' ') == 0){
         puts("You have asked to exit program; exiting ...");
         exit(0);
      }
      else{
         puts("Command not understood ... try again.");
         goto command;
      }
   }
```

RELATED FUNCTIONS

memcmp

SEE ALSO

"String Utility Functions" on page 2-3

memcpy Copy Characters

SYNOPSIS

```
#include <string.h>

void *memcpy(void *to, const void *from, size_t n);
```

DESCRIPTION

memcpy copies the number of bytes specified by **n** from one area of memory (**from**) to another (**to**). All bytes, including any null characters, are copied.

RETURN VALUE

memcpy returns a pointer to the **to** area.

CAUTION

The effect of **memcpy** when the source and target fields overlap is undefined. Sometimes a run-time diagnostic message is produced in this case.

The third argument to **memcpy** is **size_t**. If a negative number is passed, overlaying of memory may occur.

IMPLEMENTATION

The compiler generates inline code for **memcpy** unless **memcpy** is undefined (by an **#undef** statement) to prevent this. The inline code may still call a library routine in special cases (for example, if the length is a variable whose value is larger than 16 megabytes).

The code generated for **memcpy** usually uses the MVCL instruction to perform data movement. If more than 16 megabytes of data are to be moved, the library routine is called, which moves 16M-1 bytes at a time. (Thus, the effect on overlapping fields can depend on whether they are separated by as much as 16 megabytes.) For more information on optimizing your use of **memcpy**, see “Optimizing Your Use of memcmp, memcpy, and memset” on page 2-5.

memcpy Copy Characters
(continued)

EXAMPLE

```
#include <string.h>
#include <stdio.h>

#define TAILSIZE 10

main()
{
   char buf[160];
   char tail[TAILSIZE+1];

   puts("Enter a line of text.");
   gets(buf);
   if (strlen(buf) < TAILSIZE)
      printf("Your input was shorter than %d characters.\n",
             TAILSIZE);
   else{
      memcpy(tail, buf+strlen(buf)-TAILSIZE, TAILSIZE+1);

         /* Copy last 10 characters of buf, */
         /* plus the trailing null.         */
      printf("The last 10 characters of your input were "
                "\"%s\".\n", tail);
   }
}
```

RELATED FUNCTIONS

memcpyp, **memmove**, **strcpy**, **strncpy**

SEE ALSO

"String Utility Functions" on page 2-3

memcpyp Copy Characters (with Padding)

SYNOPSIS

```
#include <lcstring.h>

void *memcpyp(void *target, const void *source,
              size_t tsize, size_t ssize, int pad);
```

DESCRIPTION

memcpyp copies bytes from the **source** area to the **target** area, padding out to the target size (**tsize**) after **ssize** bytes are copied from **source**. The code generated by **memcpyp** uses the MCVL instruction to perform data movement.

RETURN VALUE

memcpyp returns a pointer to the target area.

IMPLEMENTATION

The compiler generates inline code for **memcpyp** unless **memcpyp** is undefined (by an **#undef** statement) to prevent this. The inline code may still call a library routine in special cases (for example, if the length is a variable whose value is larger than 16 megabytes).

The code generated for **memcpyp** usually uses the MVCL instruction to perform data movement. If more than 16 megabytes of data are to be moved, the library routine is called, which moves 16M-1 bytes at a time. (Thus, the effect on overlapping fields can depend on whether they are separated by as much as 16 megabytes.)

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

main()
{
   char *a;
   char name[41], input[25];

   puts("Enter your name:");
   a = gets(input);

      /* Copy input to name; pad with '*' to 40 characters. */
   memcpyp(name, a, 40,strlen(a), '*');
   a[40]='\0';
   printf("Name padded to 40 characters : %.40s\n", name);
}
```

memcpyp *(continued)*

Copy Characters (with Padding)

RELATED FUNCTIONS

`memcpy`

SEE ALSO

"String Utility Functions" on page 2-3

memfil Fill a Block of Memory with a Multicharacter String

SYNOPSIS

```
#include <lcstring.h>

void *memfil(void *to, const void *from, size_t n,
             size_t len);
```

DESCRIPTION

memfil fills a block of memory (indicated by **to**) with the contents of the array **from**. The argument **n** specifies the length of the area to be filled, while **len** specifies the size of the fill sequence. The area length does not have to be evenly divisible by the string length, in which case only a portion of the sequence is included in the final copy. Null characters in the fill sequence are treated like any other character.

RETURN VALUE

memfil returns the address of the **to** area.

CAUTION

The third argument to **memfil** has type **size_t**. If a negative number is passed, massive overlaying of memory occurs. (The fourth argument also has type **size_t**, but specification of a negative value here may produce incorrect results, but overlaying of memory will not occur.)

If the fill string and the target area overlap, the effect of **memfil** is undefined. If the string length is 0, the target area is not changed.

IMPLEMENTATION

memfil uses the MVCL instruction to propagate characters through memory. MVCL also copies the fill string to the target unless the fill string is longer than 256 bytes, in which case **memcpy** is called to do this.

EXAMPLE

```
#include <lcstring.h>

int minus2 = -2;
int values[100] [100];

   /* Set all array elements to -2. */
memfil(values, &minus2, sizeof(values), sizeof(int));
```

RELATED FUNCTIONS

memset

SEE ALSO

"String Utility Functions" on page 2-3

memlwr Translate a Memory Block to Lowercase

SYNOPSIS

```
#include <lcstring.h>

void *memlwr(void *memblk, size_t n);
```

DESCRIPTION

memlwr scans the first **n** bytes of the input memory block addressed by **memblk**, converting uppercase characters (’A’ through ’Z’) to lowercase characters (’a’ through ’z’).

memlwr is not affected by a program’s locale.

RETURN VALUE

memlwr returns a pointer to the memory block.

CAUTION

The second argument to **memlwr** is **size_t**. If a negative number is passed, massive overlaying of memory occurs.

EXAMPLE

```
#include <stdio.h>
#include <lcstring.h>

static struct PART {
   size_t length;
   char *word;
   } sentence[] = { {  4, "THIS " },
                    {  8, "EXAMPLE " },
                    {  6, "SHOWS " },
                    {  8, "THE USE " },
                    {  7, "OF THE " },
                    {  7, "memlwr " },
                    { 11, "FUNCTION.\n" } };

#define NUM_PARTS (sizeof(sentence)/sizeof(struct PART))

main()
{
   int x;

   for (x = 0; x < NUM_PARTS; x++) {
      memlwr(sentence[x].word, sentence[x].length);
      fputs(sentence[x].word, stdout);
   }
   exit(0);
}
```

memlwr *(continued)* Translate a Memory Block to Lowercase

RELATED FUNCTIONS

`memupr`, **`memxlt`**, **`strlwr`**

SEE ALSO

"String Utility Functions" on page 2-3

memmove Copy Characters

SYNOPSIS

```
#include <string.h>

void *memmove(void *to, const void *from, size_t n);
```

DESCRIPTION

memmove is identical to **memcpy** except that the copy is guaranteed to work correctly even if the **to** and **from** objects overlap. On completion of the call, the **n** bytes addressed by **to** are identical to the **n** bytes addressed by **from** before the call.

RETURN VALUE

memmove returns a pointer to the **to** area.

CAUTION

The third argument to **memmove** is **size_t**. If a negative number is passed, massive overlaying of memory may occur.

EXAMPLE

This example removes hyphens from a word by shifting text to the left to overlay any hyphens, using **memmove**:

```
#include <string.h>
#include <stdio.h>

#define MAXLEN 100

main()
{
   char *word;
   size_t len;
   char *hyphen;
   char line[MAXLEN];

   puts("Enter a hyphenated word: ");
   word = gets(line);
   printf("\noriginal word: %s\n", word);
   len = strlen(word);
   for (;;) {
       hyphen = strchr(word, '-');
       if (!hyphen) break;
       memmove(hyphen, hyphen + 1, len - (hyphen - word));
          /* Remove hyphen from word. */
       --len;
   }
   printf("Unhyphenated word: %s\n", word);
}
```

memmove *(continued)*

Copy Characters

RELATED FUNCTIONS

`memcpy`

SEE ALSO

"String Utility Functions" on page 2-3

memscan Scan a Block of Memory Using a Translate Table

SYNOPSIS

```
#include <lcstring.h>

char *memscan(const char *area, const char *table,
              size_t len);
```

DESCRIPTION

memscan scans a block of memory using a translate table. A *translate table* is an array of 256 bytes, one for each EBCDIC character. The scan terminates at the first character in the block whose table entry is not 0.

area is a pointer to the block of memory to be scanned, and **table** addresses the first byte (corresponding to the character '\0') of the translate table. **len** specifies the maximum number of bytes to be scanned. If any character in the memory block has a nonzero table entry, then the scan terminates at the first such character, and the address of that character is returned. If no character in the block has a nonzero table entry, then **memscan** returns **NULL**.

RETURN VALUE

memscan returns a pointer to the first character in the memory block whose table entry is nonzero, or **NULL** if all characters have 0 entries in the table.

IMPLEMENTATION

memscan is implemented as a built-in function and uses the TRT instruction to search for a character with a nonzero entry in the table.

EXAMPLE

See the example for **memscntb**.

RELATED FUNCTIONS

memchr, **memscntb**, **strscan**

SEE ALSO

"String Utility Functions" on page 2-3

memscntb Build a Translate Table for Use by memscan

SYNOPSIS

```
#include <lcstring.h>

char *memscntb(char *area, const char *chars,
               int val, size_t len);
```

DESCRIPTION

memscntb builds a translate table that is used with the **memscan** function to scan a block of memory for the first occurrence of one of a set of characters. A *translate table* is an array containing 1 byte for each of the 256 EBCDIC characters. **memscan** scans its argument area for a character whose value in the translate table is not 0.

area is either the address of a 256-byte array or **NULL**. If **area** is **NULL**, **memscntb** builds the translate table in a static area whose address is returned. If **area** is not **NULL**, the table is built in the specified array.

chars is an array of characters that are to be translated to the same value. The **len** argument specifies the number of characters in the **chars** array. The table byte corresponding to each character in the first **len** bytes of the **chars** array has the value specified by **val**, while all other characters have the value of **!val** (that is, 1 if val is 0, and 0 otherwise). For example, if **chars** is “ab”, **len** is 2, and **val** is 0, then bytes 129 and 130 (the EBCDIC decimal values for a and b) in the translate table have the value 0, and all other bytes have the value 1.

The null character is treated like any other character and must be present in the **chars** array if the corresponding table entry (the first byte) is to contain the value of **val**.

When building a translate table with **memscntb**, you must consider how you will use **memscan**. If you are going to search for the first occurrence of a character in **chars**, **val** should be nonzero. If you want to search for the first character *not* in **chars**, **val** should be 0.

RETURN VALUE

memscntb returns a pointer to the translate table. If **area** is **NULL**, this table cannot be modified by the program.

CAUTION

If **memscntb** is called with a **NULL area** value, the table addressed by the return value is a static area. This area may be modified by the next call to any of these functions: **memchr**, **memscntb**, **strchr**, **strcspn**, **strpbrk**, **strscntb**, and **strspn**.

IMPLEMENTATION

memscntb is implemented as a built-in function. Inline code is generated if **str** is a string literal and **val** is an integer constant.

memscntb Build a Translate Table for Use by memscan
(continued)

EXAMPLE

This example scans an area of memory for digits, and verifies that a sequence of a digit followed by a letter does not occur using **memscntb**:

```
#include <lcstring.h>
#include <stdio.h>
#include <ctype.h>

#define LEN 100

main()
{
   char area[LEN];
   char *digitp, *start;
   size_t len_left = LEN - 1;
   char *digtable;

      /* Build a translate table with nonzero entries for digits. */
   digtable = memscntb(NULL, "0123456789", 1, 10);

   for (start = area; ; start = digitp + 1) {

         /* Find next digit.                                     */
      digitp = memscan(start, digtable, len_left);
      if (!digitp) break;
      if (isalpha(*(digitp + 1))) {
         printf("Invalid sequence: %.2s\n", digitp);
         break;
      }
   }
}
```

RELATED FUNCTIONS

memscan, **strscntb**

SEE ALSO

"String Utility Functions" on page 2-3

memset Fill a Block of Memory with a Single Character

SYNOPSIS

```
#include <string.h>

void *memset(void *to, int ch, size_t n);
```

DESCRIPTION

memset fills a block of memory (indicated by **to**) with the specified character (**ch**). The size of the area to be filled is **n**.

RETURN VALUE

memset returns a pointer to the **to** area.

CAUTION

The third argument to **memset** is **size_t**. If a negative number is passed, massive overlaying of memory occurs.

IMPLEMENTATION

The compiler generates inline code for **memset** unless **memset** is undefined (by an **#undef** statement) to prevent this. The inline code can still call a library routine in special cases (for example, if the length is a variable whose value is larger than 16 megabytes).

The code generated for **memset** usually uses the MVCL instruction to propagate characters through memory. If more than 16 megabytes of memory are to be filled, the library routine is called, which processes 16M-1 bytes at a time. For more information on optimizing your use of **memcpy**, see "Optimizing Your Use of memcmp, memcpy, and memset" on page 2-5.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

main()
{
   char padded_str[65], *unpadded_str, input[66];
   int ln;

   puts("Enter a string no longer than 64 characters.");
   unpadded_str = gets(input);

      /* Copy unpadded_str to padded_str, padding with stars. */
   ln = strlen(unpadded_str);
   if (ln <= 64) {
      memcpy(padded_str, unpadded_str, ln);
      memset(padded_str +ln, '*', 64 - ln);
   }
```

memset Fill a Block of Memory with a Single Character
(continued)

```
      padded_str[64] = '\0';
      printf("The unpadded string is:\n %s\n", unpadded_str);
      printf("The padded string is :\n %s\n", padded_str);
   }
```

RELATED FUNCTIONS

`memfil`

SEE ALSO

"String Utility Functions" on page 2-3

memupr Translate a Memory Block to Uppercase

SYNOPSIS

```
#include <lcstring.h>

void *memupr(void *memblk, size_t n);
```

DESCRIPTION

memupr scans the first **n** bytes of the input memory block addressed by **memblk**, converting lowercase characters ('a' to 'z') to uppercase characters ('A' to 'Z').

memupr is not affected by a program's locale.

RETURN VALUE

memupr returns a character pointer to the memory block.

CAUTION

The second argument to **memupr** is **size_t**. If a negative number is passed, massive overlaying of memory occurs.

EXAMPLE

```
#include <stdio.h>
#include <lcstring.h>

static struct PART {
   size_t length;
   char *word;
   } sentence[] = { {  4, "This " },
                    {  8, "example " },
                    {  6, "shows " },
                    {  8, "the use " },
                    {  7, "of the " },
                    {  7, "memupr " },
                    { 11, "function.\n" } };

#define NUM_PARTS (sizeof(sentence)/sizeof(struct PART))

main()
{
   int x;

   for (x = 0; x < NUM_PARTS; x++) {
      memupr(sentence[x].word, sentence[x].length);
      fputs(sentence[x].word, stdout);
   }
   exit(0);
}
```

memupr Translate a Memory Block to Uppercase
(continued)

RELATED FUNCTIONS

`memlwr`, **`memxlt`**, **`strupr`**

SEE ALSO

"String Utility Functions" on page 2-3

memxlt Translate a Block of Memory

SYNOPSIS

```
#include <lcstring.h>

void *memxlt(void *blk, const char *table, size_t n);
```

DESCRIPTION

memxlt translates a block of memory from one character set to another. The first argument (**blk**) is the address of the area of memory to be translated, and the third argument (**n**) is the number of characters to be translated. The second argument (**table**) is a pointer to a 256-byte translate table, which should be defined so that **table[c]** for any character **c** is the value to which **c** should be translated. (The function **xltable** is frequently used to build such a table.)

Note: The argument string is translated in place; that is, each character in the string is replaced by a translated character.

RETURN VALUE

memxlt returns a pointer to the translated string.

CAUTION

The third argument to **memxlt** is **size_t**. If a negative number is passed, massive overlaying of memory occurs.

The effect of **memxlt** is not defined if the source string and the translate table overlap.

IMPLEMENTATION

If the number of bytes to be translated is a small integer constant (less than or equal to 256), the compiler generates inline code for **memxlt** unless the function is undefined (by an **#undef** statement) to prevent this.

EXAMPLE

This example produces a secret word using **memxlt**. The argument word specified on the command line is translated using a randomly arranged alphabet. The translated message is then printed to **stdout**:

```
#include <lcstring.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
   int len, i, j;
   char a;
   char alphabet[]  = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   char table[256];
```

memxlt Translate a Block of Memory
(continued)

```
      if (argc < 2) {
         puts("Specify the secret word on the command line.");
         exit(4);
      }

      len = strlen(argv[1] );
      memupr(argv[1], len);     /* Uppercase input message. */

         /* Randomize the alphabet.                         */
      for (i = 0; i < 26; i++)
         for (j = i + 1; j < 26; j++)
            if (rand() % 2) {
               a = alphabet[i];
               alphabet[i]  = alphabet[j] ;
               alphabet[j]  = a;
            }

         /* Build a translate table.                        */
      xltable(table,"ABCDEFGHIJKLMNOPQRSTUVWXYZ",alphabet);

         /* Translate message.                              */
      memxlt(argv[1],table,len);

         /* Print message.                                  */
      printf("Today's secret word is: \n%s\n",argv[1]);
      return;
   }
```

RELATED FUNCTIONS

memlwr, **memupr**, **xltable**

SEE ALSO

"String Utility Functions" on page 2-3

min Find the Minimum of Two Integers

SYNOPSIS

```
#include <lclib.h>

int min(int s, int r);
```

DESCRIPTION

min finds the minimum of two integer values, **s** and **r**.

RETURN VALUE

min returns an integer value that represents the minimum of the two arguments.

IMPLEMENTATION

min is a built-in function.

EXAMPLE

```
#include <lcmath.h>
#include <stdio.h>

main()
{
   int num1, num2; /* numbers to be compared            */
   int result;     /* holds the smaller of num1 and num2 */

   puts("Enter num1 & num2 : ");
   scanf("%d %d", &num1, &num2);
   result = min(num1, num2);
   printf("The smaller number is %d\n", result);
}
```

RELATED FUNCTIONS

max, **fmax**, **fmin**

SEE ALSO

"Mathematical Functions" on page 2-6

mkdir Create Directory

POSIX UNIX

SYNOPSIS

```
#include <sys/stat.h>

int mkdir(char *pathname, mode_t mode);
```

DESCRIPTION

mkdir creates a new OpenEdition directory and **pathname** is the directory name. **mode** is the set of file permission bits for the new directory. The owner ID of **pathname** is set to the effective user ID of the process. The group ID of **pathname** is set to the owning directory's group ID.

The pathname must specify an OpenEdition HFS filename. For programs not compiled with the **posix** option, a style prefix may be required. For more information on specifying OpenEdition filenames, see "Low-level and Standard I/O" on page 3-54.

RETURN VALUE

mkdir returns 0 if it is successful and −1 if it is not successful; it does not create a new directory.

EXAMPLE

The example for **rewinddir** also illustrates the use of **mkdir**.

RELATED FUNCTIONS

creat, **rmdir**, **umask**

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

mkfifo Create FIFO Special File

SYNOPSIS

```
#include <sys/stat.h>

int mkfifo(char *pathname, mode_t mode);
```

DESCRIPTION

`mkfifo` creates a new OpenEdition FIFO special file. **`pathname`** is the special filename, and **`mode`** is the set of file permission bits for the new FIFO file. The owner ID of **`pathname`** is set to the effective user ID of the process. The group ID of **`pathname`** is set to the owning directory's group ID.

RETURN VALUE

`mkfifo` returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

This example creates a FIFO and uses it to read and print a message from itself. Note that this example depends on the length of the message being less than the **`POSIX PIPE_BUF`** constant. A longer message causes this example to deadlock:

```
/* This program must be compiled with the posix option */

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <limits.h>

main()
{
   int rc;
   int fifofd;
   char input[_POSIX_PIPE_BUF];
   rc = mkfifo("named.pipe", S_IRUSR | S_IWUSR);
   if (rc != 0) {
      perror("mkfifo failure");
      exit(EXIT_FAILURE);
   }
   /* Open the FIFO for read. */
   fifofd = open("named.pipe", O_RDONLY);
   if (fifofd < 0){
      perror("open failure");
      remove("named.pipe");
      exit(EXIT_FAILURE);
   }
   rc = system("echo >named.pipe "
               "Talking to yourself is educational!");
```

mkfifo Create FIFO Special File
(continued)

```
      if (rc != 0) {
         fprintf(stderr, "echo failed with status code %d\n", rc);
         remove("named.pipe");
         exit(EXIT_FAILURE);
      }
      rc = read(fifofd, input, _POSIX_PIPE_BUF);
      if (rc < 0) {
         perror("read failure");
         remove("named.pipe");
         exit(EXIT_FAILURE");
      }
      puts("Something I just read:");
      fwrite(input, 1, rc, stdout);   /* Read input from the FIFO. */
      close(fifofd);
      remove("named.pipe");
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

`creat`, **`mkdir`**, **`mknod`**, **`pipe`**, **`umask`**

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

mktime Generate Encoded Time Value

SYNOPSIS

```
#include <time.h>

time_t mktime(struct tm *timeinfo);
```

DESCRIPTION

mktime converts a date and time (expressed as a **struct tm** value) into a **time_t** value. Although the **struct tm** type is convenient for input and output, it is difficult to compare or subtract time values of this type.

The **time_t** type is an alternate time format; it can be difficult to interpret directly, but it has compensating advantages. Notably, it is easy to compare two **time_t** values or to subtract them using the **difftime** function. You can use the **mktime** function to obtain a **time_t** value from a **struct tm** value; the opposite conversion is performed by the **localtime** routine.

The **timeinfo** argument is a pointer to the **struct tm** value to be converted. The value is assumed to represent local time, not Greenwich Mean Time. **mktime** is affected by time-zone information contained in the **TZ** environment variable, if it is defined.

The components of the **struct tm** value to be converted are not required to fall within their normal ranges. (See "Timing Functions" on page 2-11 for information on these ranges.) If any components of the value pointed to by **timeinfo** are out of range, they are adjusted appropriately before conversion. The **tm_wday** and **tm_yday** components of the value pointed to by **timeinfo** are ignored by **mktime** but are always set appropriately when it returns.

You can use **mktime** to perform arithmetic on dates and times. For example, to determine the date and time 1,000,000 seconds from a given time, add 1,000,000 to the seconds component of the time (**tm_sec**) and pass its address to **mktime**. On return from **mktime**, the time value is adjusted to contain accurate information.

mktime handles the **tm_isdst** field of the **timeinfo** argument differently, depending on whether the **TZ** environment variable has been set. **TZ** may be defined in three ways:

TZ is defined completely (that is, it defines both a standard time offset and a Daylight Savings Time (DST) offset).

In this case, if **tm_isdst** is zero, the time is assumed to be expressed relative to standard time; if **tm_isdst** is 1, the time is relative to DST. If **tm_isdst** is negative, the appropriate setting is determined. In either case, **tm_isdst** is set on return to indicate whether DST was in effect for the actual time specified. If **is_dst** is −1, it is possible to specify an impossible time, namely one that is skipped over in the transition between standard time and DST. In this case, **mktime** fails and returns −1.

TZ only defines a standard time offset.

In this case, if **tm_isdst** is positive, **mktime** fails and returns −1. Otherwise, standard time is assumed, and **tm_isdst** is set to 0 on return.

mktime Generate Encoded Time Value
(continued)

TZ is not defined.
In this case, it is impossible to determine whether standard time or DST is in effect. **tm_isdst** is set to −1 on return; if it was not negative on entry, a diagnostic is generated. The offset from Greenwich Mean Time is based on the assumption that the hardware TOD clock reflects Greenwich Mean Time. Except for the warning if **tm_isdst** is nonnegative, this behavior is compatible with prior releases of SAS/C.

RETURN VALUE

mktime returns an encoded time value that corresponds to the input argument. If the argument cannot be converted, −1 is returned.

CAUTION

If the argument to **mktime** specifies a value outside the range of the IBM 370 time-of-day clock (between the years 1900 and 2041), the value returned is unpredictable.

EXAMPLE

This example computes the number of days until Christmas, using **mktime**:

```
#include <time.h>
#include <stdio.h>

main()
{
   time_t now;
   static struct tm today, xmas = {0};
   time_t xmas_time;

      /* Get today's date and time (encoded).    */
   time(&now);

      /* Break into components.                  */
   today = *localtime(&now);

      /* Build midnight Christmas time structure */
      /*  for this year.                         */
   xmas.tm_mday  = 25;
   xmas.tm_mon   = 11;
   xmas.tm_year  = today.tm_year;
   xmas.tm_isdst = -1;

      /* Get encoded Christmas time.             */
   xmas_time = mktime(&xmas);

      /* Convert seconds to days and print.      */
   printf("Only %d days until Christmas.\n",
         (int)(difftime(xmas_time, now) / 86400.0));
}
```

mktime Generate Encoded Time Value
(continued)

RELATED FUNCTIONS

`localtime`, `tzset`

SEE ALSO

"Timing Functions" on page 2-11

modf Floating-Point Conversion: Fraction-Integer Split

SYNOPSIS

```
#include <math.h>

double modf(double y, double *p);
```

DESCRIPTION

`modf` separates an argument of type **`double`** into fractional and integer parts.

RETURN VALUE

`modf` returns the fractional part of the argument **`y`** with the same sign as **`y`**. The integer part of **`y`**, expressed as a floating-point number, is stored in the location referenced by the pointer **`p`**.

IMPLEMENTATION

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

EXAMPLE

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

main()
{
   double weight;
   double intweight;
   float limit = 0.5;

   puts("Enter a weight");
   scanf("%lf",&weight);
      /* Check to see if weight is negative.                  */
   if (weight < 0) {
      puts("Weight can not be a negative number");
      exit(1);
   }

      /* Test whether fractional part equals or exceeds limit.  */
   if (modf(weight, &intweight) >= limit)
       weight = intweight + 1; /* If yes, add 1 to weight.      */
   else
       weight = intweight;     /* Otherwise, round down weight. */
   printf("Your weight rounded off to the nearest pound is %f\n",
          weight);
}
```

modf Floating-Point Conversion: Fraction-Integer Split
(continued)

RELATED FUNCTIONS

`ceil`, `floor`, `fmod`

SEE ALSO

"Mathematical Functions" on page 2-6

oeddinfo Use DDname to get Information about an HFS File

SYNOPSIS

```
#include <os.h>

int oeddinfo(const char *ddnm, char path[256], unsigned *opts,
             unsigned *mode, unsigned short *disp);
```

DESCRIPTION

Under OpenEdition, **oeddinfo** returns information about a DD statement that has been allocated to an HFS file. **ddnm** is a null-terminated string that specifies the DDname. The DDname may be in either uppercase or lowercase letters. Leading white space is not permitted.

The remaining arguments are pointers that address areas in which information about the allocation is stored. Any of these pointers can be **NULL**, in which case the corresponding information is not stored.

path is an area in which **oeddinfo** stores the HFS pathname referenced by the DD statement. Names referenced by a DD statement have a limit of 256 characters.

opts is a pointer to an area where the information specified by PATHOPTS in the DD statement is stored. If you did not specify PATHOPTS, a value of 0 is stored. Each PATHOPTS keyword corresponds to an open option defined in **<fcntl.h>**. For example, the PATHOPTS keyword OAPPEND corresponds to the O_APPEND open option in **<fcntl.h>**. The PATHOPTS value stored by **oeddinfo** is the sum of the corresponding open flags. The following code tests to see if the PATHOPTS keyword OAPPEND was specified:

```
(pathopts & O_APPEND) == O_APPEND
```

Here, **&pathopts** was passed as the third argument to **oeddinfo**. To test for ORDONLY, OWRONLY or ORDWR, use the mask value O_ACCMODE. The following code tests for ORDONLY:

```
(pathopts & O_ACCMODE) == O_RDONLY
```

mode is a pointer to an unsigned **int** where information about the PATHMODE specification on the DD statement is stored. If you did not specify PATHMODE, 0 is stored. Each PATHMODE keyword corresponds to an access mode defined in **<sys/stat.h>**. For example, the PATHMODE keyword SIXOTH corresponds to the S_IXOTH access mode. The PATHMODE value stored by **oeddinfo** is the inclusive **or** of the corresponding access mode bits. The following code determines if the PATHMODE keyword SIXOTH was specified:

```
(pathmode & S_IXOTH) == S_IXOTH
```

Here, **&pathmode** was passed as the fourth argument to **oeddinfo**. Some access modes, such as S_IRWXO, are combinations of other modes; code tests for these modes carefully.

oeddinfo Use DDname to get Information about an HFS File
(continued)

`disp` is a pointer addressing an unsigned **`short`** where information about the PATHDISP specification on the DD statement is stored. If you did not specify PATHDISP, 0 is stored. The following flags are defined in **`<os.h>`** for use in testing the value stored:

NDISP_KEEP	keeps the HFS file after normal termination.
NDISP_DELETE	deletes the HFS file after normal termination.
ADISP_KEEP	keeps the HFS file after ABEND.
ADISP_DELETE	deletes the HFS file after ABEND.

RETURN VALUE

`oeddinfo` returns 0 if the DDname is defined and references an HFS file. It returns 1 if the DDname is defined but does not reference an HFS file, in which case no information is stored in any of the arguments. **`oeddinfo`** returns −1 if the DDname is not defined, or if an error occurs accessing the DD statement.

PORTABILITY

`oeddinfo` is implemented only under OpenEdition.

IMPLEMENTATION

Information about the DDname is obtained by using the information retrieval function of SVC 99.

RELATED FUNCTIONS

`osddinfo`, **`stat`**

SEE ALSO

"File Management Functions" on page 2-16

oesigsetup Control Signals

SYNOPSIS

```
#include <lcsignal.h>

int oesigsetup(sigset_t oeset *, sigset_t sascset *);
```

DESCRIPTION

oesigsetup determines which signals are managed by OpenEdition MVS and which are managed by the SAS/C library. **oesigsetup** must be called before any signal-related function other than the signal set functions, such as **sigfillset** and **sigaddset**.

If there is no call to **oesigsetup** in a program called with **exec**-linkage, the library assumes that all signals should be managed through OpenEdition, if possible.

If there is no call to **oesigsetup** in a program not called with **exec**-linkage (a regular batch or TSO execution), the library assumes that no signals should be managed by OpenEdition. You must call **oesigsetup** in a program without **exec**-linkage if you need to use OpenEdition signals.

oeset defines the set of signals to be managed by OpenEdition. **sascset** defines the signals to be managed by SAS/C. **oesigsetup** fails if a signal is included in both sets; any signal not mentioned is managed in the same way as if **oesigsetup** had not been called.

The signals that can be managed either by SAS/C or by OpenEdition, the *flexible signals*, can be divided into two groups: error signals and other signals. The error signals are

- **SIGABND**
- **SIGABRT**
- **SIGFPE**
- **SIGILL**
- **SIGSEGV**

Error signals are normally associated with program error conditions. Unless these signals are handled as OpenEdition signals, OpenEdition is not informed of the error when the corresponding error condition occurs. If the error leads to termination, OpenEdition sets the final status of the terminated process to *terminated by SIGKILL* rather than a more specific status.

The other flexible signals are listed here:

SIGALRM	If **SIGALRM** is managed by OpenEdition, the **alarmd** and **sleepd** functions are not available. If **SIGALRM** is managed by the SAS/C library, the **ps** shell command does not accurately indicate when the process is sleeping.
SIGINT	If **SIGINT** is managed by SAS/C, **SIGINT** is generated by the TSO attention key for a program running under TSO. If **SIGINT** is handled by OpenEdition, SAS/C does not use the **STAX** macro or attempt to handle TSO attentions. SAS/C management of **SIGINT** is not useful in non-TSO address spaces.

oesigsetup Control Signals
(continued)

`SIGIO`	`SIGIO` has no special meaning at present for OpenEdition or SAS/C; it may be used by future versions of either product.
`SIGTERM`	`SIGTERM` has no defined meaning to SAS/C; it can be generated only using `raise` if it is managed by SAS/C.
`SIGUSR1, SIGUSR2`	These signals have no special meaning for OpenEdition. SAS/C user-added signal support defines a meaning for one of these symbols only if they have been defined by `oesigsetup` as signals managed by SAS/C.

If you have defined a signal as managed by SAS/C and the signal is generated by OpenEdition, the result is the OpenEdition default action for the signal. For example, if you define `SIGTERM` as a signal managed by SAS/C and establish a handler, and another process uses `kill` to send your process a `SIGTERM` signal, your handler is not called, and the process is terminated.

A program can use `kill` to send a signal that `oesigsetup` has defined as a signal managed by SAS/C. If a program sends the signal to itself, only default handling takes place.

EXAMPLE

See the example for `kill`.

RELATED FUNCTIONS

`sigaddset`

SEE ALSO

"Signal-Handling Functions" on page 2-17

onjmp Define Target for Nonlocal goto

SYNOPSIS

```
#include <lcjmp.h>

void onjmp(jmp_buf env, int code, target);
```

DESCRIPTION

onjmp defines the **target** label as a point to which control should be transferred by a call to **longjmp**. The **longjmp** call specifies the same **env** value as **onjmp**. The integer value specified by the **longjmp** call is stored in the **code** integer.

onjmp sets the **code** variable to 0 on completion. Because the occurrence of a **longjmp** assigns **onjmp** a value other than 0, the value of **code** can be tested elsewhere to determine whether a jump has taken place.

RETURN VALUE

onjmp has no return value.

CAUTION

Variables of storage class **auto** and **register**, whose values are changed between the **onjmp** and **longjmp** calls, have indeterminate values after the branch to **target** unless declared **volatile**.

onjmp is implemented as a macro and should not be used in any position where a value is required.

PORTABILITY

onjmp is not a Standard construct; however, the macro that implements it is portable to any system that implements **setjmp**.

IMPLEMENTATION

onjmp is implemented as

```
#define onjmp(e, c, t) if (c = setjmp(e)) goto t
```

See **setjmp** for further implementation information.

EXAMPLE

```
#include <stdio.h>
#include <lcjmp.h>
#include <stdlib.h>

jmp_buf env;
void dummy();

main()
{
   int errcode;
```

onjmp Define Target for Nonlocal goto
(continued)

```
      /* Allow restart via longjmp. */
   onjmp(env, errcode,cleanup);
   dummy();
   puts("No error occurred in dummy routine.");
   return;
   cleanup:
   printf("Beginning cleanup for error number %d\n", errcode);
}

void dummy()
{
   puts("Entering dummy routine.");
   longjmp(env, 3);
   puts("Never reached.");
}
```

RELATED FUNCTIONS

`longjmp`, **`setjmp`**

SEE ALSO

"Program Control Functions" on page 2-9

onjmpout Intercept Nonlocal gotos

SYNOPSIS

```
#include <lcjmp.h>

void onjmpout(jmp_buf env, int code, target);
```

DESCRIPTION

onjmpout requests interception of calls to **longjmp** that could terminate the calling function. If a call to **longjmp** is later intercepted, control is passed to the **target** label. The **env** and **code** variables are modified to indicate the target and code specified by the intercepted **longjmp** so that it can be reissued by the intercepting routine.

onjmpout sets the **code** variable to 0 on completion. Because interception of a jump assigns the **code** variable a value other than 0, it can be tested elsewhere to determine whether interception has taken place.

After a call to **longjmp** is intercepted, **onjmpout** must be reissued if you want continued interception.

Because **exit** is implemented as a **longjmp** to the caller of **main**, you use **onjmpout** to intercept program exit.

RETURN VALUE

onjmpout has no return value.

CAUTION

Variables of storage class **auto** and **register**, whose values are changed between the **onjmpout** and **longjmp** calls, have indeterminate values after the branch to **target** unless declared **volatile**.

onjmpout is implemented as a macro and should not be used in any position where a value is required.

IMPLEMENTATION

onjmpout is implemented as

```
#define onjmpout(e, c, t) if (c = blkjmp(e)) goto t
```

EXAMPLE

The following code fragment illustrates the use of **onjmpout**:

```
#include <lcjmp.h>

jmp_buf env;
int jmpcode;

   /* Intercept abnormal exits.                      */
onjmpout(env, jmpcode, cleanup);
.
.
.
```

onjmpout Intercept Nonlocal gotos
(continued)

```
cleanup:   /* Clean up after attempted exit.         */
.
.
.
longjmp(env, jmpcode); /* And then reissue the jump. */
```

RELATED FUNCTIONS

blkjmp, **longjmp**

SEE ALSO

"Program Control Functions" on page 2-9

open Open a File for UNIX Style I/O

SYNOPSIS

```
#include <fcntl.h>

int open(const char *name, int flags, mode_t mode);
```

DESCRIPTION

open opens a file for UNIX style I/O.

The **name** argument is the name of the file. The **flags** argument to **open** is a bit string formed by ORing option bits. The bits are defined symbolically; the header file **fcntl.h** should be included to obtain their definitions. The flags and their meanings are as follows:

O_RDONLY	opens for reading only.
O_WRONLY	opens for writing only.
O_RDWR	opens for both reading and writing.
O_APPEND	seeks to end of file before each write.
O_CREAT	creates a new file if it does not exist.
O_TRUNC	discards old data from existing file.
O_EXCL	does not accept an existing file.
O_TEXT	opens for text access.
O_BINARY	opens for binary access.
O_NOCTTY	does not make the terminal the controlling terminal of the process. This flag applies only to OpenEdition HFS files.
O_NONBLOCK	For an OpenEdition FIFO, does not delay completion of the **open** call until another process opens the FIFO. For other HFS files that support **O_NONBLOCK**, allows **read** and **write** calls to return immediately if no input is available. This flag only applies to OpenEdition HFS files.

Only one of the **O_RDONLY**, **O_WRONLY**, or **O_RDWR** flags should be set. **O_EXCL** is ignored if **O_CREAT** is also not set. If neither **O_TEXT** nor **O_BINARY** is specified, **O_BINARY** is assumed, unless the file to be opened is the terminal.

Note: The **O_TEXT** and **O_BINARY** flags have no effect when you open an OpenEdition HFS file.

The **mode** argument is optional and must be specified only when the **flags** argument specifies **O_CREAT** and **name** specifies an OpenEdition HFS file. In this case, **mode** indicates the permissions with which the file is created. For any other file type, or if the file to be opened already exists, the **mode** argument is ignored.

See "Opening Files" on page 3-27 for more information on the filename and open mode specifications.

RETURN VALUE

open returns the file descriptor if the file was successfully opened. Otherwise, it returns −1.

open Open a File for UNIX Style I/O
(continued)

PORTABILITY

File numbers for OpenEdition files are allocated using the same strategy as that used by traditional UNIX compilers; that is, the smallest available file number is always used. File numbers for files other than OpenEdition files opened by **open** are assigned very large file numbers to avoid consuming file numbers in the range supported by OpenEdition.

IMPLEMENTATION

UNIX style I/O is implemented for files that do not have suitable attributes for the **''rel''** access method by copying the file contents to a temporary file and copying modified data back when the file is closed. See "SAS/C I/O Concepts" on page 3-14 for further implementation information.

Standard I/O permits, at most, 256 files to be open at one time, including the three standard files. When UNIX style I/O is used with a non-HFS file whose attributes are not suitable for **''rel''** access, two file numbers are needed, one for the file specified by the program and one for a temporary file to which data are copied. For this reason, you might be limited to as few as 126 simultaneously-open, UNIX style files.

EXAMPLE

```
#include <fcntl.h>

main()
{
   int datafile;
   datafile = open("MYDATA", O_WRONLY|O_CREAT|O_EXCL,S_IRUSR|S_IWUSR);
   if (datafile == -1)
      puts("Unable to create MYDATA");
}
```

RELATED FUNCTIONS

aopen, **cmsopen**, **fopen**, **opendir**, **osbopen**, **osopen**, **umask**

SEE ALSO

"Opening Files" on page 3-27

"UNIX style I/O" on page 3-17

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"I/O Functions" on page 2-13

_open Open an HFS File for I/O

DESCRIPTION

`_open` is a version of `open` designed to operate only on HFS files. `_open` runs faster and calls fewer other library routines than `open`. Refer to `open` for a full description. `_open` is used exactly like the standard `open` function. The first argument to `_open` is interpreted as an HFS filename, even if it appears to begin with a style prefix or a leading // or both.

opendir Open Directory

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(char *dirname);
```

DESCRIPTION

`opendir` opens an OpenEdition directory to be read by **`readdir`**. The **`dirname`** function is the name of the directory to be read. The **`dirname`** must specify the name of a file in the OpenEdition hierarchical file system. For information on specifying OpenEdition filenames, see “File Naming Conventions” on page 3-56.

RETURN VALUE

If it is successful, **`opendir`** returns a pointer to a DIR object that describes the directory. If it is unsuccessful, **`opendir`** returns a **`NULL`** pointer.

EXAMPLE

The example for **`rewinddir`** also demonstrates the use of the **`opendir`** function.

RELATED FUNCTIONS

`cmsdfind`, **`open`**, **`osdfind`**,**`readdir`**, **`rewinddir`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

osddinfo Use DDname to get Information about a Data Set

SYNOPSIS

```
#include <os.h>

int osddinfo(char *ddnm, char dsnm[45], char member[9],
             char *recfmp, int *lreclp, int *blksizep);
```

DESCRIPTION

osddinfo obtains and returns information about the data set referenced by a particular DDname. The first argument, **ddnm**, is a null-terminated string, specifying the DDname of the data set. The DDname may be in uppercase or lowercase letters, and leading white space is not permitted.

The remaining arguments to **osddinfo** are pointers that address areas where the data set information is stored. Any of these pointers can be **NULL** , which causes the corresponding information not to be stored. Because **osddinfo** obtains only the requested information and some information is time consuming to obtain, you should always pass **NULL** to **osddinfo** for any information that you do not need.

The data set name is stored in a 45-character array that is addressed by the argument **dsnm**. Any trailing blanks are removed from the name. If the DDname is allocated to something other than a disk data set, a null (0-length) string is stored in **dsnm**.

A PDS member is stored in a 9-character array that is addressed by the argument **member**. If the DDname does not define a PDS member, a null string is stored.

A number of flags are stored in a single character that is addressed by the argument **recfmp**; the flags describe the record format of the file. The symbolic values for these flags can be found in the header file **<os.h>**:

RECFM_F	indicates fixed length records.
RECFM_V	indicates variable length records.
RECFM_U	indicates undefined length records.
RECFM_D	indicates variable length ASCII records.
RECFM_T	indicates track overflow.
RECFM_B	indicates blocked records.
RECFM_S	indicates spanned/standard records.
RECFM_A	indicates ANSI defined control characters.
RECFM_M	indicates machine control characters.

Note: You should test for **RECFM_U** before testing for **RECFM_F** or **RECFM_V** because the definition of **RECFM_U** is

```
RECFM_U = RECFM_F | RECFM_V
```

A 0 is stored in ***recfmp** if record format information is not available.

osddinfo Use DDname to get Information about a Data Set
(continued)

The data set's logical-record length is stored in an integer addressed by the argument **lreclp**. If the logical-record length is not defined or cannot be obtained, 0 is stored. If the data set is defined with **LRECL=X**, the special value **LRECL_X** is stored.

The data set's block size is stored in an integer addressed by the argument **blksizep**. If the block size is not defined or cannot be obtained, 0 is stored.

If a DD statement is defined as an HFS file, the information normally returned by **osddinfo** does not apply. If the DDname passed to **osddinfo** references an HFS file, no return information is stored; **osddinfo** returns the integer 1. You can call **oeddinfo** for the same DDname to extract information about the HFS file.

RETURN VALUE

osddinfo returns 0 if the DDname is defined, 1 if the DDname references an HFS file, or −1 if it is not defined.

IMPLEMENTATION

Information about the DDname is obtained by using the RDJFCB macro. If necessary, the OBTAIN macro is also used.

EXAMPLE

This example allocates a buffer for an input file. The buffer size should be the **lrecl** if the file is **F** or **U** format, or the **lrecl-4** if the file is V format:

```
#include <os.h>
#include <stdlib.h>

char recfm;
int lrecl;
char *buffer;

   if (osddinfo("SYSIN", NULL, NULL, &recfm, &lrecl, NULL) == 0)
      if (lrecl != 0 && lrecl != LRECL_X)
         buffer = malloc(recfm & RECFM_F? lrecl: lrecl - 4);
.
.
.
```

RELATED FUNCTIONS

cmsstat, **fattr**, **oeddinfo**, **osdsinfo**, **stat**

SEE ALSO

"File Management Functions" on page 2-16

osdfind Find the First MVS File or Member Matching a Pattern

SYNOPSIS

```
#include <os.h>

int osdfind(struct DSNINFO *info, const char *pattern);
```

DESCRIPTION

osdfind searches for an MVS data set or PDS member whose name matches the pattern pointed to by **pattern**. Information about the first matching file is returned in the structure pointed to by **info**. Additional matching files can be retrieved using the **osdnext** function.

The pattern has the form of a MVS data set name, with several extensions:

- □ If the pattern ends with an asterisk in parentheses (*), only PDS members are considered to match the pattern. If any nonpartitioned files match the rest of the pattern, they are ignored.
- □ If the pattern begins with a period, it is prefixed by the user's TSO prefix or by the userid for batch jobs. For example, if the current TSO prefix is MARY, then .C.OBJ(*) would match the same files as MARY.C.OBJ(*).
- □ If the pattern contains an asterisk as a qualifier, the pattern matches any data set name that replaces the asterisk with a single qualifier. For example, FRED.*.C would match FRED.UPDATE.C and FRED.PLAY.C, but not FRED.WORK.CODE.C or FRED.WORK.C.CODE. Only a single asterisk qualifier is allowed in a pattern. Also, the asterisk must not be specified as the first qualifier of a pattern.
- □ If the pattern ends with a period, it matches any data set with an initial segment matching the pattern without the ".". For example, FRED.*.C. would match not only FRED.UPDATE.C and FRED.PLAY.C, but also FRED.WORK.C.CODE.

The header file **<os.h>** defines the structure **DSNINFO**. This structure is used for **osdfind**, **osdnext**, and **osdquit**. The structure is defined as

```
struct DSNINFO {    /* argument structure for osdfind/osdnext  */
  void * _ [4] ;               /* reserved for library use     */
  char cattype;                /* catalog entry type           */
  char dsname [45] ;           /* null-terminated full dsname  */
  char mem [9] ;               /* null-terminated member name  */
  char __;                     /* padding                      */
  unsigned TTR: 24;            /* TTR of start of member       */
  unsigned alias: 1;           /* alias member bit             */
  unsigned user_TTR_count: 2;  /* number of user TTRs          */
  unsigned user_data_halfwords: 5;   /* halfwords of user data */
  char user_data [62] ;        /* user data from PDS directory */
};
```

The following paragraphs describe some of the members of this structure.

The **cattype** member is a one-character code indicating the type of file. The values are

A	non VSAM data set
B	GDG base
C	VSAM cluster

osdfind Find the First MVS File or Member Matching a Pattern
(continued)

D	VSAM data component
G	VSAM alternate index
I	VSAM index component
M	master catalog
P	page space
R	VSAM path
U	user catalog
X	alias name

The **mem** element is the member name when you specify a pattern ending in an asterisk in parentheses (*). The **mem** element is a null string for other patterns.

The **TTR, alias, user_TTR_count, user_data_halfwords**, and **user_data** members contain information from the PDS directory entry for the member when an (*) pattern is specified. See the *MVS Data Administration Guide* for more information on PDS directory structures. These fields are not meaningful when the pattern does not request the return of PDS members.

RETURN VALUES

osdfind returns 0 if a matching data set or PDS member is found or −1 if no match is found. Other negative values may be returned to indicate fatal errors, after which no more names can be returned. If a minor error occurs, a positive value may be returned, indicating that the names of some files or members could not be returned, but that another call to **osdfind** may successfully return additional names. For example, if **osdfind** is unable to return PDS members because a PDS directory is corrupt, **osdfind** returns a positive return code to indicate that it may be possible to return members of other PDSs matching the pattern.

CAUTION

osdfind assumes the PDS directory for an (*) pattern is not modified during processing. If a write to the data set or member occurs during the execution of **osdfind**, the results are unpredictable. The results are also unpredictable if files or members matching the pattern are added or deleted between the call to **osdfind** and the corresponding **osdquit**.

The first 16 bytes of the **DSNINFO** structure are reserved for use in subsequent calls to **osdnext**. Do not alter this area because the effects are unpredictable.

The catalog searches by **osdfind** use a 65-K buffer to hold data set names. For this reason, all matching filenames may not be returned if the highest level qualifier identifies more than about 1,400 filenames. For example, if there are more than 1,400 files under the index ORDERS, then **osdfind** with the pattern ORDERS.*.LOG returns only matching names from the first 1,400. This is a limitation of the operating system, not of SAS/C.

osdfind Find the First MVS File or Member Matching a Pattern
(continued)

EXAMPLE

This example uses **osdfind** and **osdnext** to search all the user's .C files for members that are defined in the SPF statistics as belonging to a specific userid:

```
#include <os.h>
#include <string.h>
#include <stdio.h>

void findmine(char *id) {

   struct spfstat {  /* This structure maps SPF statistics. */
      char version;
      char mod;
      char pad1[2];
      int created;       /* Julian date, packed decimal     */
      int changed;       /* Julian date, packed decimal     */
      char changet[2];   /* time changed, hours and minutes */
      short size;
      short init;
      short modified;
      char userid[8];    /* who edited it last              */
   } *s;
   struct DSNINFO d;     /* Return information from         */
                         /*  osdfind/osdnext.               */
   int rc;
   rc = osdfind(&d, ".*.c(*)");    /* Find all my .C files. */
   while(rc >= 0) {
      char resp;
      if (rc > 0) {
         puts("Error searching .C files.  Continue?");
         scanf(" %c", &resp);
         if (resp != 'y' && resp != 'Y') {
            osdquit(&d); /* If not continuing, free memory. */
            break;
         }
      }
      else if (d.user_data_halfwords >=

            /* if user data looks like SPF data             */
         sizeof(struct spfstat)/2) {
         s = (struct spfstats *) d.user_data;

            /* if it's owned by this id                     */
         if (memcmp(s->userid, id, 8) == 0)

               /* Print filename and member.                */
            printf("%s(%s)\n", d.dsname, d.mem);
      }
      rc = osdnext(&d);            /* Proceed to next one. */
   }
}
```

osdfind *(continued)* Find the First MVS File or Member Matching a Pattern

RELATED FUNCTIONS

`cmsdfind`, **`cmsffind`**, **`opendir`**, **`osdnext`**, **`osdquit`**

SEE ALSO

"File Management Functions" on page 2-16

osdnext Find the Next MVS File or Member Matching a Pattern

SYNOPSIS

```
#include <os.h>

int osdnext(struct DSNINFO *info);
```

DESCRIPTION

osdnext finds the next MVS data set or member matching a pattern previously defined in a call to **osdfind**. The **info** pointer points to a **DSNINFO** structure filled in by a previous call to **osdfind**. If a match is found, information about the matching data set or member is placed in the structure pointed to by **info**, as with **osdfind**. Refer to the description of **osdfind** for a listing of the **DSNINFO** structure.

RETURN VALUES

osdnext returns 0 if a matching data set or PDS member is found, or −1 if no match is found. Other negative values may be returned to indicate fatal errors, after which no more names can be returned. If a minor error occurs, a positive value may be returned, indicating that the names of some files or members could not be returned, but that another call to **osdnext** may successfully return additional names. For example, if **osdnext** is unable to return PDS members because a PDS directory is corrupt, **osdnext** returns a positive return code to indicate that it may be possible to return members of other PDSs matching the pattern.

CAUTION

osdnext assumes that the PDS directory for a (*) pattern will not be modified during processing. If a write to the data set or member occurs during the execution of **osdnext**, the results are unpredictable. The results are also unpredictable if files or members matching the pattern are added or deleted between the call to **osdnext** and the corresponding **osdquit**.

The first 16 bytes of the **DSNINFO** structure are reserved for use in subsequent calls to **osdnext**. Do not alter this area because the effects are unpredictable.

The catalog searches by **osdnext** use a 65K buffer to hold data set names. For this reason, all matching filenames may not be returned if the highest level qualifier identifies more than about 1,400 filenames. For example, if there are more than 1,400 files under the index ORDERS, then **osdnext** with the pattern ORDERS.*.LOG returns only matching names from the first 1,400. This is a limitation of the operating system, not of SAS/C.

EXAMPLE

See the example for **osdfind**.

osdnext Find the Next MVS File or Member Matching a Pattern
(continued)

RELATED FUNCTIONS

`osdfind`

SEE ALSO

"File Management Functions" on page 2-16

osdquit Terminate MVS File or Member Search

SYNOPSIS

```
#include <os.h>

void osdquit(struct DSNINFO *info);
```

DESCRIPTION

`osdquit` is called to free resources associated with the use of the **`osdfind`** and **`osdnext`** functions. **`info`** is a pointer to a **`DSNINFO`** structure filled in by a previous call to **`osdfind`**. Refer to the description of **`osdfind`** for a listing of the **`DSNINFO`** structure.

It is not necessary to call **`osdquit`** after **`osdfind`** or **`osdnext`** when they return a negative return code. However, **`osdquit`** is required only when **`osdfind`** or **`osdnext`** indicates that more names are available, and the program does not need to retrieve these names.

RETURN VALUE

None.

CAUTION

Refer to the CAUTION section of **`osdfind`**.

EXAMPLE

See the example for **`osdfind`**.

RELATED FUNCTIONS

`osdfind`

SEE ALSO

"File Management Functions" on page 2-16

osdsinfo Obtain Information about a Data Set by DSname

SYNOPSIS

```
#include <os.h>

int osdsinfo(const char *dsnm, int tsoform, unsigned short *dsorgp,
             char *recfmp, int *lreclp, int *blksizep);
```

DESCRIPTION

The **osdsinfo** function obtains and returns information about the data set referenced by a particular DSname. This function works only under MVS. The first argument, **dsnm**, is a null-terminated string specifying the DSname of the data set. You can specify in either uppercase or lowercase letters, and leading white space is not permitted. The **tsoform** flag indicates whether the DSname is a **tso**-style name or is fully qualified. If the value of **tsoform** is 0, the name is assumed to be complete. If the value of **tsoform** is not 0, the name is completed by prepending your TSO prefix (or your userid, if the program is running in batch and your userid can be determined).

The remaining arguments to **osdsinfo** are pointers that address areas where the data set information is stored. Any of these pointers can be **NULL**, which causes the corresponding information not to be stored. Because **osdsinfo** obtains only the requested information (and some information is time consuming to obtain), you should always pass **NULL** to **osdsinfo** for any information that you do not need. A number of flags are stored in the **unsigned short** addressed by the argument **dsorgp**; the flags describe the file's data set organization. The symbolic values for these flags can be found in the header file **<os.h>**:

```
#define DSORG_IS 0x8000 /* indexed sequential       */
#define DSORG_PS 0x4000 /* physical sequential      */
#define DSORG_DA 0x2000 /* direct organization      */
#define DSORG_PO 0x0200 /* partitioned organization */
#define DSORG_U  0x0100 /* unmovable                */
#define DSORG_AM 0x0008 /* VSAM                     */
```

A 0 is stored in ***dsorgp** if the data set organization is not available.

A number of flags are stored in a single character that is addressed by the argument **recfmp**; the flags describe the record format of the file. The symbolic values for these flags also can be found in the header file **<os.h>**:

```
#define RECFM_F 0x80   /* fixed-length records           */
#define RECFM_V 0x40   /* variable-length records        */
#define RECFM_U 0xc0   /* undefined-length records       */
#define RECFM_D 0x20   /* variable-length ASCII records  */
#define RECFM_T 0x20   /* track overflow                 */
#define RECFM_B 0x10   /* blocked records                */
#define RECFM_S 0x08   /* spanned/standard records       */
#define RECFM_A 0x04   /* ANSI-defined control characters */
#define RECFM_M 0x02   /* machine control characters     */
```

osdsinfo Obtain Information about a Data Set by DSname
(continued)

Note: You should test for **`RECFM_U`** before testing for **`RECFM_F`** or **`RECFM_V`** because the definition of **`RECFM_U`** is

```
RECFM_U = RECFM_F | RECFM_V
```

A 0 is stored in **`*recfmp`** if record format information is not available.

The data set's logical-record length is stored in an integer that is addressed by the **`lreclp`** argument. If the logical-record length is not defined or cannot be obtained, 0 is stored. If the data set is defined with **`LRECL=X`**, the special value **`LRECL_X`** is stored.

The data set's block size is stored in an integer that is addressed by the **`blksizep`** argument. If the block size is not defined or cannot be obtained, 0 is stored.

RETURN VALUE

The **`osdsinfo`** function returns 0 if information about the DSname is available, or a nonzero value if it fails. If an error occurs, the return code is the same as the error code stored in **`errno`**.

CAUTION

You should not call the **`osdsinfo`** function for a file that is already open. Such calls can fail with dynamic-allocation errors, due to interference by system I/O processing. Consider using the **`fattr`** function to get information about the attributes of a file that is already open.

PORTABILITY

`osdsinfo` is implemented only under MVS.

IMPLEMENTATION

The requested file is dynamically allocated using SVC 99. Additional information about the file that is not provided by SVC 99 is obtained by issuing the RDJFCB and OBTAIN macros.

EXAMPLE

This code fragment allocates a buffer for an input file. The buffer size should be **`lrecl`** if the file is **`F`** or **`U`** format, or the **`lrecl - 4`** if the file is V format.

```
#include <os.h>
#include <stdlib.h>

char recfm;
int lrecl;
char *buffer;
if (osdsinfo("input.data", 1, NULL, &recfm, &lrecl, NULL) == 0)
   if (lrecl != 0 && lrecl != LRECL_X)
      buffer = malloc(recfm & RECFM_F? lrecl: lrecl - 4);
.
.
.
```

osdsinfo Obtain Information about a Data Set by DSname
(continued)

RELATED FUNCTIONS

`cmsstat`, `fattr`, `osddinfo`, `stat`

SEE ALSO

"File Management Functions" on page 2-16

oslink Call an MVS Utility Program

SYNOPSIS

```
#include <os.h>

int oslink(const char *pgm, arg1, arg2 ...);
```

DESCRIPTION

oslink calls another load module (usually an MVS utility program), passing one or more arguments. The load module, whose name is specified by **pgm**, must be in the same library as the C program (in STEPLIB) or the link list or link-pack area. MVS standard linkage passes the arguments to the program; the high-order bit is set to indicate the last argument value in the list. (To avoid confusion produced by this bit, all program arguments should be defined as pointers.)

Most MVS utilities expect to receive two arguments, an option list, and a DDname list. See EXAMPLE for an example of this usage.

RETURN VALUE

The completion code returned by the called load module is returned by **oslink**. If the program cannot be called successfully, a negative value is returned. The negative return codes have the same meanings as the negative return codes from the **system** function.

CAUTION

oslink is used primarily to invoke MVS utility programs. Do not use it in place of **loadm** to dynamically load and execute C subordinate load modules because this produces an ABEND of the called module. You can use **oslink** to invoke a C **main** program, but invocation is more easily performed using the **system** function.

When **oslink** is used in MVS/XA or MVS/ESA by a program that runs in 31-bit addressing mode to call a program that runs in 24-bit addressing mode, all the arguments must be addressable using a 24-bit address. If the arguments are automatic, this is always true.

If a TSO attention interrupt occurs during a call to **oslink**, the called program is immediately terminated, unless the program handles the attention itself.

PORTABILITY

oslink is implemented only under MVS.

IMPLEMENTATION

oslink uses the MVS ATTACH macro to call the requested load module. This macro permits the C program to continue execution if the called program terminates abnormally.

EXAMPLE

This example for **oslink** invokes the IBM PL/I compiler, passing a list of options and a list of alternate DDnames. See the *PL/I Programmer's Guide* for further information.

oslink Call an MVS Utility Program
(continued)

```
#include <os.h>
#include <string.h>

   /* PL/I compiler options structure                  */
struct {
   short optlen;
   char options[100];
   }  pli_opts;

  /* PL/I DDname list                                  */
struct {
   short ddsize;
   char ddnames[8][8];
   }  pli_ddns;

int plirc;                /* PL/I compiler return code */

   /* Build options parm.                              */
strcpy(pli_opts.options, "SOURCE,NEST,XREF");
pli_opts.optlen = strlen(pli_opts.options);

   /* Store size of DDnames.                           */
memset(pli_ddns.ddnames, '\0', sizeof(pli_ddns.ddnames));
pli_ddns.ddsize = sizeof(pli_ddns.ddnames);

   /* Set alternate names. */
memcpy(pli_ddns.ddnames[0], "ALTLIN  ", 8);
memcpy(pli_ddns.ddnames[3], "ALTLIB  ", 8);
memcpy(pli_ddns.ddnames[4], "ALTIN   ", 8);
memcpy(pli_ddns.ddnames[5], "ALTPRINT", 8);
memcpy(pli_ddns.ddnames[6], "ALTPUNCH", 8);
memcpy(pli_ddns.ddnames[7], "ALTUT1  ", 8);

   /* Call compiler; save return code.                 */
plirc = oslink("IEL0AA", &pli_opts, &pli_ddns);
```

RELATED FUNCTIONS

popen, **system**

SEE ALSO

Chapter 4, “Environment Variables” on page 4-1

“System Interface and Environment Variables” on page 2-17

palloc Allocate an Element from a Storage Pool

SYNOPSIS

```
#include <pool.h>

void *palloc(POOL_t *p);
```

DESCRIPTION

`palloc` allocates an element from a storage pool created by the `pool` function and returns its address. `p` is a pointer (of type `POOL_t`) that points to the storage pool.

RETURN VALUE

`palloc` returns the address of the allocated element. `NULL` is returned if all elements are allocated and it is impossible to extend the pool.

ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

EXAMPLE

See the example for `pool`.

RELATED FUNCTIONS

`malloc`, `pool`

SEE ALSO

"Memory Allocation Functions" on page 2-10

pause Suspend Execution until a Signal Is Received

SYNOPSIS

```
#include <lcsignal.h>

int pause(void);
```

The synopsis for the POSIX implementation is

```
#include <sys/types.h>
#include <unistd.h>

int pause(void);
```

You may use either set of header files in your program.

DESCRIPTION

pause suspends program execution until a signal is discovered. If a signal occurs that is blocked, program execution does not resume. If an unblocked signal occurs, **pause** calls the signal handler, if any, and then immediately returns to its caller.

The **pause** function may be implemented by OpenEdition or internally by SAS/C. If **oesigsetup** has been called, explicitly or implicitly, then **pause** is implemented by OpenEdition; otherwise, it is implemented by SAS/C.

No CPU time is consumed (other than set-up time) while **pause** is executing.

RETURN VALUE

pause returns −1, which indicates that it was interrupted by a signal.

CAUTION

A similar function, **sigsuspend**, allows precise control of signal blocking while pausing, so it is usually preferable to **pause**.

PORTABILITY

pause is compatible with Berkeley UNIX, except the return value is different.

EXAMPLE

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <setjmp.h>
#include <lcjmp.h>
#include <stdio.h>

void breakout(int);
jmp_buf jbuf;
int jcode;

main()
```

pause Suspend Execution until a Signal Is Received
(continued)

```
{
      /* Establish SIGINT handling.    */
   onjmp(jbuf, jcode, done);
   signal(SIGINT, &breakout);
   puts("We are now pausing for a message from our sponsor.");
   puts("Enter Control C or attn to continue.");
   pause();

done:
   puts("And now back to our program.");
   return;
}

   /* SIGINT handler gets out of wait. */
void breakout(int signum)
{
   puts("Try SAS/C today, the choice of the new generation!");
   longjmp(jbuf, 1);
}
```

RELATED FUNCTIONS

`ecbpause`, **`ecbsuspend`**, **`sigpause`**, **`sigsuspend`**, **`sleep`**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

pclose Close Pipe I/O To a Process

SYNOPSIS

```
#include <stdio.h>

int pclose(FILE *pipe);
```

DESCRIPTION

pclose closes a pipe connected to a shell command that was opened by **popen**. The argument **pipe** is the FILE pointer returned by **popen**. The **pclose** function waits for the associated process to terminate.

An appropriate feature test macro (**_SASC_POSIX_SOURCE** or **_POSIX_C_SOURCE**) must be defined to make the declaration of **pclose** in **<stdio.h>** visible.

RETURN VALUE

pclose returns the exit status of the command. **pclose** returns −1 if the stream is not associated with a command called by **popen**, or if an error occurs closing the pipe.

PORTABILITY

pclose is defined in accordance with POSIX 1003.2.

EXAMPLE

See the example for **popen**.

RELATED FUNCTIONS

fclose, **popen**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

pdel Delete a Storage Pool

SYNOPSIS

```
#include <pool.h>

void pdel(POOL_t *p);
```

DESCRIPTION

`pdel` releases all elements of a storage pool (allocated by the **`pool`** function) and frees associated storage. **`p`** is a pointer that points to the storage pool. It is not necessary to free all the elements of the pool before deleting it.

RETURN VALUE

`pdel` 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.

CAUTION

Deleting a storage pool twice or calling **`pdel`** with a random argument usually results in an ABEND.

Do not reference pool elements after the pool has been deleted.

EXAMPLE

See the example for **`pool`**.

RELATED FUNCTIONS

`free`, **`pool`**

SEE ALSO

"Memory Allocation Functions" on page 2-10

pdset Packed Decimal Conversion: Double to Packed Decimal

SYNOPSIS

```
#include <packed.h>

void pdset(char (*val)[], double input,
           unsigned int scale, double round);
```

DESCRIPTION

pdset converts **input** to its packed-decimal representation. **input** is the **double** value to be converted. **val** is a pointer to a character array in which the packed decimal result is stored. The maximum size of **val** is 8 bytes (15 decimal digits). **input** is multiplied by **pow(10.0, scale)** before conversion. **scale**, which specifies a scaling factor, must be a positive integer less than or equal to 15. **round** is an amount added to **input** after scaling and before conversion. After **round** is added, any fractional portion is discarded.

RETURN VALUE

pdset has no return value.

CAUTION

If the **input** value is the result of computations with nonintegral data, a **round** value of 0 is not recommended because it can cause the effect of a small inaccuracy to be considerably magnified. For example, with a **scale** value of 2 and a **round** of 0, a computed value of 1.1699998 is stored as 116 (rather than 117).

DIAGNOSTICS

If an error occurs, **pdset** sets the location pointed to by **val** to all 9s (in packed-decimal format and with the appropriate sign) and sets **errno** to one of three values:

- If the size of the **input** field is less than 8, **errno** is set to EUSAGE.
- If **scale** is not less than or equal to 15, **errno** is set to EARG.
- If the value of **input** after scaling is too large to be converted, **errno** is set to ERANGE.

IMPLEMENTATION

pdset is defined in **<packed.h>** as

```
#define pdset(val, input, scale, round)\
   _pdset(val, sizeof(*(val)), input, scale, round)
```

pdset Packed Decimal Conversion: Double to Packed Decimal
(continued)

EXAMPLE

```
#include <packed.h>

pdstruct {
   char income[6];
   char outgo[6];

   /* expected COBOL data declarations: */
   /* INCOME PIC 9(9)V99 COMP-3.        */
   /* OUTGO  PIC 9(9)V99 COMP-3.        */
} struct;

void percent3(struct pdstruct *data)
{
   double cents;
   cents = pdval(&data->income, 0);
   cents *= 0.03;    /* Compute 3 percent. */

      /* Store in record after rounding.   */
   pdset(&data->outgo, cents, 0, 0.5);
   return;
}
```

RELATED FUNCTIONS

`pdval`

SEE ALSO

SAS/C Compiler Interlanguage Communication Feature User's Guide

"General Utility Functions" on page 2-9

pdval Packed Decimal Conversion: Packed Decimal to Double

SYNOPSIS

```
#include <packed.h>

double pdval(const char (*val)[], unsigned int scale);
```

DESCRIPTION

`pdval` converts **`val`**, a number in packed-decimal format, to its floating-point representation. The maximum length of **`val`** is 8 bytes (15 decimal digits). After conversion, **`val`** is multiplied by **`pow(10.0, -scale)`**. **`scale`**, which specifies a scaling factor, must be a nonnegative integer less than or equal to 15.

RETURN VALUE

The return value is the double-precision, floating-point representation of **`val`**, appropriately scaled.

ERRORS

If **`val`** does not contain valid packed-decimal data, an 0C7 ABEND results.

DIAGNOSTICS

If **`scale`** is not positive and less than or equal to 15, then **`pdval`** returns **`HUGE_VAL`** and sets **`errno`** to EARG.

IMPLEMENTATION

`pdval` is defined in **`<packed.h>`** as

```
#define pdval(val, scale) _pdval(val, sizeof(*val), scale)
```

EXAMPLE

```
#include <packed.h>
#include <stdio.h>

void printamt(char (*amount)[6])

   /* expected COBOL data declaration: */
   /* AMOUNT PIC 9(9)V99 COMP-3.       */
{
   double dollars;

      /* Convert to dollars and cents. */
   dollars = pdval(amount, 2);
   printf("Amount is $ % 12.2f\n", dollars);
   return;
}
```

RELATED FUNCTIONS

`pdset`

pdval Packed Decimal Conversion: Packed Decimal to Double
(continued)

SEE ALSO

SAS/C Compiler Interlanguage Communication Feature User's Guide

"General Utility Functions" on page 2-9

perror Write Diagnostic Message

SYNOPSIS

```
#include <stdio.h>

void perror(const char *prefix);
```

DESCRIPTION

perror writes a diagnostic message to the standard error file, **stderr**. The message is preceded by the **prefix** string, a colon, and a blank. It is followed by a new-line character.

The text of the message is based on the value of the external integer **errno**, which is set by the library when an error or warning condition is detected.

Note: If the library writes a diagnostic for an error condition, this message is usually more precise than the message that would be written by **perror**. This is because, in many cases, one value for **errno** corresponds to a number of different conditions, and other information about the error (for example, the name of a file) is unavailable to **perror**.

The texts of the messages and the precise meanings of the possible **errno** values are implementation dependent. The texts and explanations of the SAS/C library messages are provided in the *SAS/C Software Diagnostic Messages, First Edition.*

RETURN VALUE

perror has no return value.

EXAMPLE

```
#include <stdio.h>
#include <errno.h>

main()
{
   FILE *f;
   quiet(1);           /* Suppress library messages. */
   f = fopen("myfile","r");

   if (!f)
      if (errno == ENFOUND) fprintf(stderr,
          "Error in input phase: myfile not found.\n");
      else perror("error in input phase");

   quiet(0);                /* Allow messages again. */
}
```

RELATED FUNCTIONS

quiet, **strerror**

perror *(continued)* Write Diagnostic Message

SEE ALSO

"The errno Variable" on page 1-7

"Diagnostic Control Functions" on page 2-11

pfree Return an Allocated Element to a Storage Pool

SYNOPSIS

```
#include <pool.h>

void pfree(POOL_t *p, void *elt);
```

DESCRIPTION

`pfree` returns a previously allocated element to a storage pool (created by the **`pool`** function). **`p`** is a pointer to the pool, and **`elt`** is a pointer to the element to be returned.

RETURN VALUE

`pfree` has no return value.

CAUTION

If the returned element is not allocated from the storage pool, the results are unpredictable.

You should not reference pool elements after the pool has been freed.

EXAMPLE

See the example for **`pool`**.

RELATED FUNCTIONS

`free`, **`palloc`**, **`pool`**,

SEE ALSO

"Memory Allocation Functions" on page 2-10

pipe Create Unnamed Pipe

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

int pipe(int fd[2]);
```

DESCRIPTION

pipe creates an I/O channel through which a process communicates with another process or with itself. **fd** points to a memory area where two file descriptors are stored. **pipe** stores the file descriptor for the output end of the pipe in **fd[0]**, and it stores the file descriptor for the input end of the pipe in **fd[2]**. The first data written to the pipe are the first to be read. **O_NONBLOCK** and **FD_CLOEXEC** are turned off at both ends of the pipe.

RETURN VALUE

pipe returns 0 if it is successful, and −1 if it is not successful.

EXAMPLE

This example invokes the **ls** shell command using **fork** and **exec**, and uses a pipe allocated to file descriptor 1 to obtain the output of **ls** and write it to **stderr** (which may be a non-HFS terminal or disk file if the example is run under MVS batch or TSO):

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <lclib.h>

static void do_ls(char * const []);

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.   */
   pipe(pipefds);       /* Create both ends of a pipe.          */

      /* Make write end of pipe fd 1.                           */
   dup2(pipefds[1],STDOUT_FILENO);

      /* Close write end.                                       */
   if (pipefds[1] != 1) close(pipefds[1]);
```

pipe Create Unnamed Pipe

(continued)

```
      /* In child process, invoke ls.                            */
   if ((pid = fork()) == 0) do_ls(parmList);
   close(1);             /* 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));
}

static void do_ls(char *const parmList[]) {
   int rc;
   rc = execvp("ls", parmList);       /* Pass control to ls.     */
      /* execvp must have failed!                                */
   perror("execvp failure");
   abort();              /* Terminate process the hard way.      */
}
```

RELATED FUNCTIONS

mkfifo

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

pool Allocate a Storage Pool

SYNOPSIS

```
#include <pool.h>

int pool(POOL_t *p, unsigned eltsize, unsigned initial,
         unsigned extend);
```

DESCRIPTION

pool creates a storage pool from which elements of a given size can be quickly allocated and freed. The arguments are as follows:

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 **pool** 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 first argument to **pool**.

ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

DIAGNOSTICS

The pool pointer is set to 0 if there is no storage available for the new pool.

IMPLEMENTATION

If an initial size is not specified for a storage pool, space is allocated for seven or more elements, rounded up to fill an integral number of pages.

Under an XA or ESA operating system, memory allocated by **pool** resides above the 16-megabyte line for programs that run in 31-bit addressing mode.

The initial pool storage, as well as additional storage for extending the pool, is obtained by calling **malloc**. The performance of **palloc** and **pfree** is improved because the frequency of calls to **malloc** and **free** is thereby reduced.

EXAMPLE

```
#include <stdio.h>
#include <string.h>
```

pool Allocate a Storage Pool
(continued)

```
#include <stdlib.h>
#include <pool.h>

#define WORD_LENGTH 24
#define BUFFER_LENGTH 255
#define DELIMITERS " +-*/&=!%^*'~.,;:|?\"'[]<>()\a\b\f\n\r\t\v"

typedef struct tagLINE {
   struct tagLINE *next;
   int number;
   int count;
   } line_t;

typedef struct tagWORD{
   struct tagWORD *left, *right;
   line_t *first_line, *last_line;
   char token[WORD_LENGTH+1];
   } word_t;

static POOL_t word_pool;
static POOL_t line_pool;
static word_t *word_tree = NULL;

static word_t *alloc_word(const char *);
static line_t *alloc_line(int);
static void add_to_xref(const char *, int);
static word_t *find_word(word_t *, word_t *);
static void add_line(word_t *, int);
static void add_word(word_t **, word_t *);
static void print_xref(word_t *);
static void print_word(word_t *);

   /* Read a file of text, such as this file, from stdin.  Separate   */
   /*  each line of input into tokens.  Write a sorted                */
   /*  "cross-reference" of the file to stdout, having one line for   */
   /*  each distinct token.  For each token, show the line number(s)  */
   /*  on which the token is found.  If the token appears more than   */
   /*  once on a line, show the number of times it appears on that    */
   /*  line.                                                          */
   /*                                                                 */
   /* Here's a sample of the output produced by this program, using   */
   /*  this file as input:                                            */
   /*                                                                 */
   /*  stdin                 : 36, 46, 84, 85(2), 93, 96, 97          */
   /*  stdio                 : 1, 47                                  */
   /*  stdlib                : 3, 48                                  */
   /*  stdout                : 38, 49, 295, 304, 305                  */
   /*  storage               : 50, 107                                */

main()
{
   int success;
```

pool Allocate a Storage Pool
(continued)

```
      int line_number;
      char input_buffer[BUFFER_LENGTH+2];

         /* Allocate a pool of binary tree elements */
         /* to hold the "words". */
      success = pool(&word_pool, sizeof(word_t), 100, 100);
      if (!success) {
         puts("Can't allocate word pool.");
         exit(4);
      }

         /* Allocate a pool of list elements to hold the */
         /* line numbers.   */
      success = pool(&line_pool, sizeof(line_t), 500, 250);
      if (!success) {
         puts("Can't allocate line count pool.");
         exit(4);
      }

        /* Read each line in the input file.  Pick out tokens */
        /* and add them to the cross-reference tree.          */
      line_number = 0;
      fgets(input_buffer, BUFFER_LENGTH, stdin);
      while (!feof(stdin) && !ferror(stdin)) {
         char *token;
         line_number += 1;
         token = strtok(input_buffer, DELIMITERS);
         while (token) {
            add_to_xref(token, line_number);
            token = strtok(NULL, DELIMITERS);
         }
         fgets(input_buffer, BUFFER_LENGTH, stdin);
      }

      if (ferror(stdin)) {
         puts("Error reading stdin.");
         exit(8);
      }

         /* Print the cross-reference, one word per line.     */
      print_xref(word_tree);

         /* Free the storage pools and exit.                  */
      pdel(&word_pool);
      pdel(&line_pool);

      exit(0);
   }

      /* Allocate a new word_t element from the word pool.              */
      /*  Initialize all members and save a copy of the token,          */
      /*  truncating the token if it's longer than WORD_LENGTH.         */
```

pool Allocate a Storage Pool

(continued)

```
   /*  Exit with an error message if palloc fails.                      */
static word_t *alloc_word(const char *token)
{
   word_t *new_word;

   new_word = (word_t *) palloc(&word_pool);
   if (new_word == NULL) {
      puts("Can't allocate element from word pool");
      exit(12);
   }

   new_word->left = NULL;
   new_word->right = NULL;
   new_word->first_line = NULL;
   new_word->last_line = NULL;
   strncpy(new_word->token, token, WORD_LENGTH);
   new_word->token[WORD_LENGTH] = '\0';

   return new_word;
}

   /* Allocate a new line_t element from the line pool.  Initialize    */
   /*  all members.   Exit with an error message if palloc fails.      */
static line_t *alloc_line(int line_number)
{
   line_t *new_line;

   new_line = (line_t *) palloc(&line_pool);
   if (new_line == NULL) {
      puts("Can't allocate element from line pool");
      exit(12);
   }

   new_line->next = NULL;
   new_line->number = line_number;
   new_line->count = 1;

   return new_line;
}

   /* Add this instance of the word to the cross-reference tree.       */
static void add_to_xref(const char *token, int line_number)
{
   word_t *word, *new_word;

      /* Go ahead and copy the token to a word_t element.              */
   new_word = alloc_word(token);

      /* If the word is already in the tree, free the      */
      /* word_t element we just allocated and add the      */
      /* line number to the word_t element we found.       */
```

pool Allocate a Storage Pool
(continued)

```
      /* Otherwise, add the new word_t element to the     */
      /* tree and add this line number.                   */
   word = find_word(word_tree, new_word);
   if (word != NULL) {
      pfree(&word_pool, new_word);
      add_line(word, line_number);
   }
   else {
      add_word(&word_tree, new_word);
      add_line(new_word, line_number);
   }

}

   /* Search for the word in the word binary tree.        */
   /* Return NULL if the word is not on the tree.         */
static word_t *find_word(word_t *subtree, word_t *new_word)
{
   word_t *word;

   if (subtree == NULL)
      return NULL;

   word = find_word(subtree->left, new_word);
   if (word != NULL)
      return word;

   if (strcmp(subtree->token, new_word->token) == 0)
      return subtree;

   word = find_word(subtree->right, new_word);
   if (word != NULL)
      return word;

   return NULL;
}

   /* Add an instance of the word for this line.  If      */
   /* the word has already been used on this line,        */
   /* simply increment the count for this line number.    */
   /* Otherwise add a new line_t element for this line    */
   /* number.                                             */
static void add_line(word_t *word, int line_number)
{
   line_t *line;

   if (word->last_line != NULL &&
                          word->last_line->number == line_number)
      word->last_line->count += 1;
```

pool Allocate a Storage Pool
(continued)

```
   else {
      line = alloc_line(line_number);
      if (word->first_line == NULL)
         word->first_line = line;
      else
         word->last_line->next = line;
      word->last_line = line;
   }

}

   /* Add the new word_t element to the binary tree.                  */
static void add_word(word_t **subtree, word_t *new_word)
{
   int relation;

   if (*subtree == NULL) {
      *subtree = new_word;
      return;
   }

   relation = strcmp((*subtree)->token, new_word->token);
   if (relation > 0)
      add_word(&((*subtree)->left), new_word);
   else if (relation < 0)
      add_word(&((*subtree)->right), new_word);
   else
      abort();                                /* impossible condition */

}

   /* Print a list of the words in alphabetical order.  Beside each   */
   /*  word print the line numbers on which the word appears.  If the */
   /*  word appears more than once on a line, print the line number   */
   /*  followed by a repeat count in parentheses.  For example, if    */
   /*  the word appears twice on line number 20, print "20(2)".       */
static void print_xref(word_t *subtree)
{

   if (subtree != NULL) {
      print_xref(subtree->left);
      print_word(subtree);
      print_xref(subtree->right);
   }

}
```

pool Allocate a Storage Pool
(continued)

```
   /* Print the line number info for a single word.                  */
static void print_word(word_t *word)
{
   line_t *line;
   char comma;

   printf("%-24s: ", word->token);

   comma = 0;
   for (line = word->first_line; line != NULL; line = line->next) {
      if (comma)
         fputs(", ", stdout);
      comma = 1;
      printf("%d", line->number);
      if (line->count > 1)
         printf("(%d)", line->count);
   }

   putchar('\n');

   if (ferror(stdout)) {
      puts("Error writing to stdout.");
      exit(12);
   }
}
```

RELATED FUNCTIONS

malloc, **palloc**, **pdel**, **pfree**

SEE ALSO

"Memory Allocation Functions" on page 2-10

popen Open Pipe I/O to a Shell Command

POSIX UNIX

SYNOPSIS

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

DESCRIPTION

popen creates a pipe between the calling program and a command to be executed by the OpenEdition shell. A stream opened by **popen** should be closed by **pclose**.

The arguments are pointers to null-terminated strings. **command** is a null-terminated shell command. **mode** is the I/O mode, which can be set to these values:

r indicates read mode. You read from the standard output of the command by reading from the **FILE** pointer returned by **popen**.

w indicates write mode. You write to the standard input of the command by writing to the **FILE** pointer returned by **popen**.

Because open files are shared, you can use a mode of **''r''** as an input filter and a mode of **''w''** as an output filter.

You must define an appropriate feature test macro (**_SASC_POSIX_SOURCE** or **_POSIX_C_SOURCE**) to make the declaration of **popen** in **<stdio.h>** visible.

Note: A stream opened by **popen** must be closed by **pclose**.

RETURN VALUE

popen returns a **FILE** pointer if successful. **popen** returns a NULL pointer if a file or process cannot be created.

PORTABILITY

popen is defined in accordance with POSIX 1003.2

EXAMPLE

This example sorts the lines of an HFS file and writes out the first line of the sorted file. The **popen** function is used to invoke the shell sort command to do all the work.

```
/* This program must be compiled with the posix compiler option */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINE 500

main(int argc, char *argv[])
{
   char linebuf[MAXLINE];
   char *cmdbuf;
   int cmdlen;
   FILE *sort_output;
```

popen Open Pipe I/O to a Shell Command
(continued)

```
      if (argc < 2) cmdbuf = "sort";
      else {
         if (argc > 2) fputs("Extraneous arguments ignored\n", stderr);
         cmdlen = 5+strlen(argv[1]);

            /* Allocate space for sort command.                    */
         cmdbuf = malloc(cmdlen);
         if (!cmdbuf) exit(EXIT_FAILURE);
         sprintf(cmdbuf, "sort %s", argv[1]); /* Build sort command.*/
      }
      sort_output = popen(cmdbuf, "r");  /* Read the output of sort.*/
      if (!sort_output) {
         perror("popen failure");
         exit(EXIT_FAILURE);
      }

         /* Read first sorted line.                                */
      fgets(linebuf, sizeof(linebuf), sort_output);

      if (feof(sort_output) || ferror(sort_output)) {
         fputs("Input error.\n");
         pclose(sort_output); /* Close sort process before quitting.*/
         exit(EXIT_FAILURE);
      }
      puts(linebuf);          /* Write line to stdout.             */

         /* Close the sort process. It will probably terminate      */
         /*  with SIGPIPE.                                          */
      pclose(sort_output);
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

pclose, **pipe**, **system**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

pow Compute the Value of the Power Function

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double pow(double x, double y);
```

DESCRIPTION

pow computes the value of **x** raised to the power **y**, as expressed by this relation:

$$r = x^y$$

RETURN VALUE

pow returns the value of its argument **x** raised to the power **y**. The result is a double-precision, floating-point number.

DIAGNOSTICS

If $\mathbf{x}^{\mathbf{y}}$ is too large to be represented, the run-time library writes an error message to the standard error file (**stderr**) and returns ± **HUGE_VAL**. If $\mathbf{x}^{\mathbf{y}}$ is too small to be represented, the run-time library writes an error message to the standard error file (**stderr**) and returns 0.0.

For a negative value of **x** and a noninteger **y**, the function returns 0.0, and the run-time library writes an error message to **stderr**. For **x == 0.0** and negative **y**, the function returns **HUGE_VAL**, and the run-time library writes an error message to **stderr**.

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

EXAMPLE

This example finds the cube root of 17, using **pow**:

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

main()
{
   double x, y, f;

   x = 17.0;
   y = 1.0/3.0;
   f = pow(x, y);
   printf("(pow(%f,%f)) = %f\n", x, y, f);
}
```

pow *(continued)* Compute the Value of the Power Function

RELATED FUNCTIONS

`exp`, **`_matherr`**

SEE ALSO

"Mathematical Functions" on page 2-6

printf Write Formatted Output to the Standard Output Stream

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, var1, var2, ...);
```

DESCRIPTION

printf writes output to the standard output stream under the control of the string addressed by **format**. In the argument list following **format**, there may be one or more additional arguments whose values are to be formatted and transmitted.

The string pointed to by **format** is in the same form as that used by **fprintf**. Refer to the **fprintf** description for detailed information concerning the formatting conversions.

RETURN VALUE

printf returns the number of characters transmitted to **stdout**.

DIAGNOSTICS

If there is an error during output, **printf** returns a negative value.

IMPLEMENTATION

printf is identical to **fprintf** with **stdout** as the output file.

EXAMPLE

This example displays a number of integer and floating-point values using different **printf** formats to contrast the behavior of these formats:

```
#include <stdio.h>

int values[] = {
  0, 25, 1048576, -1, 6000};
double fvalues[] = {
  13, 55555.5, .00034562, 14.99999816, -6.37e11};

main()
{
   int i;

      /* Label the output columns.                          */
   printf("Integral formats:\n%-15s%-15s%-15s%-15s%-15s\n\n",
          "%d", "%+.5d", "%u", "%#o", "%x");

      /* Note: All formats include a "-15" specification to   */
      /*  force them to appear in 15 columns, left-justified. */
   for(i = 0; i < sizeof(values)/sizeof(int); ++i)
      printf("%-15d%+-15.5d%-15u%-#15o%-15x\n",
             values[i],values[i],values[i],values[i],
             values[i]);
```

printf Write Formatted Output to the Standard Output Stream
(continued)

```
        printf("\nFloating-point formats:
               \n%-16s%-16s%-16s%-16s%-16s\n\n",
              "%.5e", "%.8e", "%.8g", "%#.8g", "%.5f");

           /* Note: All formats include a "-16" specification to   */
           /*  force them to appear in 16 columns, left-justified. */
        for(i = 0; i < sizeof(fvalues)/sizeof(double); ++i)
           printf("%-16.5e%-16.8e%-16.8g%-#16.8g%-16.5f\n",
                  fvalues[i],fvalues[i],fvalues[i],fvalues[i],
                  fvalues[i]);
     }
```

RELATED FUNCTIONS

fprintf, **sprintf**, **vprintf**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

putc Write a Character to a File

SYNOPSIS

```
#include <stdio.h>

int putc(int c, FILE *f);
```

DESCRIPTION

putc writes a single character **c** to the stream associated with the **FILE** object addressed by **f**.

RETURN VALUE

putc returns the output character or **EOF** if an error occurs.

IMPLEMENTATION

putc is implemented as a built-in function. A subroutine call is executed only if no output buffer is allocated, the output buffer is full, or a control character is written.

The code generated for **putc** normally includes tests for a NULL **FILE** pointer and for a stream that failed to open. If you define the symbol **_FASTIO** using **#define** or the **DEFine** compiler option before including **<stdio.h>**, no code is generated for these checks. This enables you to improve the performance of debugged programs that use **putc**.

EXAMPLE

This example copies characters from an input file to an output file, and it writes a blank after each period:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define _FASTIO  /* Improve getc/putc performance. */

main()
{
   FILE *infile, *outfile;
   char filename[60];
   int c;

   puts("Enter the name of your input file:");
   memcpy(filename, "tso:", 4);
   gets(filename+4);
   infile = fopen(filename, "r");
   if (!infile){
      puts("Failed to open input file.");
      exit(EXIT_FAILURE);
   }
   puts("Enter the name of your output file:");
   memcpy(filename, "tso:", 4);
   gets(filename+4);
```

putc Write a Character to a File
(continued)

```
         outfile = fopen(filename, "w");
         if (!outfile){
            puts("Failed to open output file.");
            exit(EXIT_FAILURE);
         }
         for (;;){
            c = getc(infile);
            if (c == EOF) break;
            c = putc(c, outfile);
            if (c == '.') putc(' ', outfile);
         }

         fclose(infile);
         fclose(outfile);
      }
```

RELATED FUNCTIONS

fputc, **putchar**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

putchar Write a Character to the Standard Output Stream

ISO ANSI UNIX

SYNOPSIS

```
#include <stdio.h>

int putchar(int c);
```

DESCRIPTION

`putchar` writes a character **`c`** to the stream **`stdout`**.

RETURN VALUE

`putchar` returns the character written or **`EOF`** if an error occurs.

EXAMPLE

This example writes the first line of a file to **`stdout`**:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

main()
{
   int c;
   FILE *infile;
   char filename[60];

   puts("Enter the name of your input file:");
   memcpy(filename, "tso:", 4);
   gets(filename+4);
   infile = fopen(filename, "r");
   if (!infile){
      puts("Failed to open input file.");
      exit(EXIT_FAILURE);
   }

      /* While character is not a newline character,  */
      /*  read character from file MYFILE.            */
   while (((c = getc(infile)) != '\n') && (c != EOF))

      /* Write one character of the file to the       */
      /*  standard output; this line is written one   */
      /*  character at a time.                        */
   putchar(c);
   putchar('/n');
}
```

putchar Write a Character to the Standard Output Stream
(continued)

RELATED FUNCTIONS

`putc`

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

putenv Modify or Define Environment Variable

SYNOPSIS

```
#include <lclib.h>

int putenv(const char *string);
```

DESCRIPTION

The **putenv** function creates an environment variable with a specified name and value, or replaces an existing environment variable with a new value. The name and value are specified by the string pointed to by the argument **string**. For example,

[[*scope*]:][[*groupname*].]*varname*[=*value*]

For portable use, the *scope* and *groupname* parts of the string must be omitted, and a *value* must be present. The parts are

scope
: specifies the scope at which the environment variable is to be added or modified. It may be specified as one of the following:

PRogram

External (or STorage)

PErmanent (or Lasting)

SEssion.

scope is not case sensitive. The uppercase letters indicate the minimum abbreviation that may be specified for the *scope* name. See Chapter 4, "Environment Variables" on page 4-1 for a definition of the environment-variable scopes. The SEssion scope is CMS specific and refers to GLOBALV SESSION variables. For all other systems, a SEssion-scope specification is treated as if it were an External-scope specification. If you do not specify scope, PRogram scope is assumed. Scopes other than PRogram are valid only under TSO, CMS, and CICS.

groupname
: specifies an optional group name for the environment variable. *groupname* is only meaningful for nonprogram-scope variables. If a program-scope, environment-variable name appears to have a group name, the group name is simply treated as part of the variable name. See Chapter 4, "Environment Variables" on page 4-1 for more information.

varname
: specifies the name of the environment variable. For nonprogram-scope variables, some environments restrict the size of the variable name. In these environments, the name is truncated, if necessary.

=*value*
: specifies the value to be assigned to the environment variable. If no value is present, a 0-length string `""` is assumed. For nonprogram-scope variables, the value is truncated if it is longer than supported for that scope.

putenv Modify or Define Environment Variable
(continued)

RETURN VALUE

putenv returns 0 if successful or −1 if unsuccessful.

CAUTION

Do not modify the environment by changing the external variable **environ** or the data it points to in a program that uses **putenv**. The **putenv** function may cause the value of **environ** to change.

PORTABILITY

putenv is defined by many UNIX and MS-DOS C compilers. Scopes and groups for environment variables are SAS/C extensions.

USAGE NOTES

You can define the same variable name in more than one scope. However, **getenv** always returns the value of shortest duration. For example, if a program-scope variable is defined, **getenv** always returns its value.

EXAMPLE

This example creates an environment variable named **HOME**, if it does not already exist, and then invokes an OpenEdition shell command:

```
#include <stdio.h>
#include <lclib.h>
#include <lcstring.h>

main()
{
   char home[17];
   char cmd[300];
   int rc;

      /* if environment variable HOME not defined               */
   if (!getenv("HOME")) {
      strcpy(home, "HOME=/u/");
      cuserid(home+8);      /* Append userid to directory name.  */
      strlwr(home);         /* Translate to lowercase letters.   */
      rc = putenv(home);    /* Define HOME environment variable. */
      if (rc != 0) {
         perror("putenv failure");
         exit(EXIT_FAILURE);
      }
   }
   puts("Enter shell command");
   memcpy(cmd, "//sh:", 5); /* prefix for system function        */
   gets(cmd+5);
   rc = system(cmd);        /* Invoke the shell command.         */
   printf("shell command status code was %d.\n", rc);
   exit(rc);
}
```

putenv Modify or Define Environment Variable
(continued)

RELATED FUNCTIONS

`clearenv`, **`getenv`**, **`setenv`**

SEE ALSO

Chapter 6, “Executing C Programs,” in *SAS/C Compiler and Library User’s Guide*

Chapter 4, “Environment Variables” on page 4-1

“System Interface and Environment Variables” on page 2-17

puts Write a String to the Standard Output Stream

SYNOPSIS

```
#include <stdio.h>

int puts(const char *str);
```

DESCRIPTION

puts(str) is equivalent to **fputs(str, stdout)**, except that a new-line character is written to **stdout** after the last character of **str**.

RETURN VALUE

puts returns EOF if an error occurs. Otherwise, **puts** returns a nonzero value.

EXAMPLE

This example writes the following two lines to **stdout**

- the first line of example output.
- the second line of example output, written in two pieces.

```
#include <stdio.h>

main()
{
   puts("The first line of example output.");
   fputs("The second line of example output ",stdout);
   puts("written in two pieces.");
}
```

RELATED FUNCTIONS

afwrite, **fputs**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

qsort Sort an Array of Elements

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *blk, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

DESCRIPTION

`qsort` sorts the array pointed to by `blk`, using the quicksort algorithm. `n` is the number of elements in the array. `size` is the element size in bytes. `cmp` is a user-provided comparison function.

`qsort` calls `cmp` with pointers to two elements of the array. `cmp` determines which element is larger or whether the two elements are equal. The `cmp` function thereby defines the ordering relation for the elements to be sorted. The precise comparison technique that should be implemented by the `cmp` function depends on the type of data to be compared and on the application. A typical comparison function is illustrated under EXAMPLE below.

`cmp` returns these values:

- a negative integer, if the first of the two elements is less than the second
- a positive integer, if the first element is greater than the second
- 0, if the two elements are equal.

If the two elements are equal, their order after sorting is arbitrary.

RETURN VALUE

`qsort` has no return value.

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>

#define MAXEMPLOY 100
#define EMPLOYEE_FILENAME "employee"

typedef struct {            /* Define employee record. */
   unsigned employ_no;
   char last_name[30];
}  employee;
employee emp_tabl[MAXEMPLOY];

   /* Compare function for bsearch and qsort.          */
static int compare_employees(const void *, const void *);

main()
{
   FILE *employ_file;
   int employ_count = 0;
   unsigned srchid;     /* Search value (employee id). */
   char *temp_emp;
```

qsort Sort an Array of Elements
(continued)

```
   employ_file = fopen(EMPLOYEE_FILENAME, "rb");

      /* Error checking omitted.                      */
      /* Read in employee file.                       */
   while (!feof(employ_file)) {
      fread(&emp_tabl[employ_count], sizeof(employee),
            1, employ_file);
      ++employ_count;
   }
   fclose(employ_file);
      /* Sort employee table by employee number.      */
   qsort(emp_tabl, employ_count, sizeof(employee),
         &compare_employees);

   puts("Enter Employee ID to Search for:");
   scanf("%d", &srchid);        /* Enter search data. */

      /* Do a lookup with bsearch for an entry in the  */
      /*  employee table. It uses the same             */
      /*  compare_employees function.                  */
   temp_emp = bsearch(&srchid, emp_tabl, employ_count,
                sizeof(employee), &compare_employees);
   if (temp_emp == NULL)   /* Print results of search. */
      printf("Invalid ID\n");
   else
      printf("Last Name: %s\n",
            ((employee *) temp_emp)->last_name);
}

static int compare_employees(const void *first,const void *second)
{
   employee *efirst, *esecond;

   efirst  = (employee *)first;
   esecond = (employee *)second;

      /* Return -1 if first emp no < second emp, 0 if  */
      /* they are equal, or 1 if first > second.       */
   return (efirst->employ_no > esecond->employ_no) -
          (efirst->employ_no < esecond->employ_no);
}
```

RELATED FUNCTIONS

bsearch

SEE ALSO

"General Utility Functions" on page 2-9

quiet Control Library Diagnostic Output

SYNOPSIS

```
#include <lclib.h>

void quiet(int flag);
```

DESCRIPTION

`quiet` suppresses library diagnostics or allows them to be produced. A nonzero argument requests message suppression, and a 0 argument allows diagnostics to be printed (if they have not been suppressed by other active functions).

`quiet` operates on a function-by-function basis. If any active function calls **`quiet(1)`** and does not reset it by calling **`quiet(0)`**, diagnostics are suppressed. (Thus, a function cannot countermand a **`quiet(1)`** issued by its caller. Also note that **`quiet(1)`** has no effect after the function that calls it has terminated.)

You can suppress diagnostics for the entire execution of a program if **`main`** issues **`quiet(1)`** during start-up. If you want to suppress messages in coprocesses, be sure that the initial **`main`** function within each coprocess contains a call to **`quiet`**. Calling the **`quiet`** function within the **`main`** function of a program that subsequently creates coprocesses does not suppress messages within the new coprocess. See Chapter 9, "Coprocessing Functions" in the *SAS/C Library Reference, Volume 2* for more information on coprocesses.

If the library option **`=warning`** is specified at run time, calls to **`quiet`** have no effect. (See Chapter 8, "Run-Time Argument Processing," in the *SAS/C Compiler and Library User's Guide* for more information about **`=warning`**.)

RETURN VALUE

`quiet` has no return value.

CAUTION

You should use **`quiet`** only when a message would be issued by the library for a harmless condition, or in which the caller can generate a more informative message for all possible warning conditions.

EXAMPLE

```
#include <lclib.h>
#include <stdio.h>

main()
{
   FILE *f;
   quiet(1);                      /* Suppress library messages. */
   f = fopen("ddn:SYSLIN", "r"); /* Try to open SYSLIN.       */
   quiet(0);                      /* Allow messages again,      */
   if (!f){                       /* or use SYSIN as a backup.  */
```

quiet Control Library Diagnostic Output
(continued)

```
            puts("Unable to open SYSLIN, using SYSIN instead.");
            f = fopen("ddn:SYSIN", "r");
         }
      }
```

RELATED FUNCTIONS

`perror`

SEE ALSO

"Diagnostic Control Functions" on page 2-11

raise Generate an Artificial Signal

SYNOPSIS

```
#include <signal.h>

int raise(int signum);
```

DESCRIPTION

raise is called to artificially generate a signal. The argument **signum** is the number of the signal to be generated. You should specify this argument symbolically, using one of the signal names listed in "Types of Signal Support" on page 5-2 . When **raise** is called, the signal is handled as established by a previous call to **signal** or by the default handler if **signal** was not called.

If the **raise** function is called for a signal managed by OpenEdition, **raise** calls the **kill** function to generate the signal. Then, if the signal is blocked when **raise** is called, the signal remains pending until the signal is unblocked. If a signal managed by SAS/C is generated with **raise**, the handler is always called immediately, regardless of whether the signal is asynchronous or blocked. If you call **siginfo** in a handler for a signal generated by **raise**, it always returns **NULL**.

RETURN VALUE

raise returns 0 if it is successful or a nonzero value if it is not successful. The most common reason that **raise** fails is that the **signum** number is not a valid signal.

Note: **raise** may not return if the handler for the signal terminates with a call to **longjmp**, **exit**, or **abort**, or if the handler for the signal defaults to abnormal program termination.

CAUTION

If you use **raise** to test signal handling in a program, keep in mind that signals generated by **raise** are treated somewhat differently from signals that occur naturally. Signals other than OpenEdition signals that are generated by **raise** are always handled immediately, so you should not use **raise** to test signal blocking. You may also encounter problems if you use **raise** to test the handler for an asynchronous signal because the handler for the signal is executed immediately, not just when a function is called or returns.

EXAMPLE

Refer to the example for **siginfo**.

RELATED FUNCTIONS

kill, **siggen**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

rand Simple Random Number Generation

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
```

DESCRIPTION

rand returns pseudorandom numbers in the range from 0 to **RAND_MAX**. **RAND_MAX** is defined as 32767 in **<stdlib.h>**. The sequence of pseudorandom numbers is controlled by the value of **seed**. You can set this value by a call to **srand**. You can call **srand** at any time to reset the number generator to a new starting point. The initial default **seed** is 1.

RETURN VALUE

rand returns a random number between 0 and 32767.

PORTABILITY

The exact sequence of generated values for a particular **seed** and the exact range in which values can be generated may vary from implementation to implementation. (The sequence of numbers produced for a given **seed** by the library is the same as the usual UNIX C library implementation.)

The algorithm used for **rand** in this implementation is described in the ANSI Standard. The period is 2^{32} calls. Because the value **v** returned is in the range $0 \leq v \leq 32767$, individual values of **v** may be repeated after about 2^{16} calls, but the sequence as a whole does not repeat until 2^{32} calls.

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>

main()
{
   char card, suit;

      /* sets seed to 22                         */
   srand (22);

      /* Assign a random value to card and suit. */
   card = "A23456789TJQK"[rand()%13];
   suit = "CDHS"[rand()%4];

   printf("Your card: %c %c\n", card, suit);
}
```

rand Simple Random Number Generation
(continued)

RELATED FUNCTIONS

`srand`

SEE ALSO

"Mathematical Functions" on page 2-6

read Read Data from a File or a Socket

POSIX UNIX

SYNOPSIS

```
#include <fcntl.h>

int read(int fn, void *buf, unsigned size);
```

The synopsis for the POSIX implementation is

```
#include <sys/types.h>
#include <unistd.h>

ssize_t read(int fn, void *buf, size_t size);
```

DESCRIPTION

read reads data from the file or socket with file descriptor **fn** into the buffer addressed by **buf**. At most, **size** bytes are read. If **size** is 0, **read** returns a value of 0 and does not attempt any other operation. If **fn** is associated with a socket, it must either be connected or have been assigned an associated remote address by the **connect** function.

RETURN VALUE

read returns the number of bytes read if it is successful. A returned 0 indicates that the end of file has been reached, and a returned −1 indicates a failure. Note that for sockets, terminal files, and OpenEdition special files, it is not an error if fewer bytes are read than requested. Also, for HFS files or sockets that have been defined as nonblocking by the **fcntl** function, a return value of 0 indicates that no data were immediately available and does not necessarily indicate end of file.

The remaining information in this section applies when you use **read** to read an OpenEdition HFS file. If **read** is interrupted by a signal, it returns a −1 if it has not read any data; otherwise, it returns the number of bytes read before the interruption. **read** returns 0 if the starting position is at or beyond the end of the file.

If **read** attempts to operate on an empty regular file or FIFO special file, it returns 0 if no process has the pipe open for writing. **read** returns −1 if a process has the pipe open for writing and **O_NONBLOCK** is set to 1. **read** does not return until data are written or until the pipe is closed by all other processes if a process has the pipe open for writing and **O_NONBLOCK** is set to 0.

CAUTION

read is an atomic operation. When using User Datagram Protocol (UDP), no more than one datagram can be read per call. If you are using datagram sockets, make sure there is enough buffer space to contain an incoming datagram.

EXAMPLE

This example appends a copy of an MVS file to itself. Because it accesses the file as binary, the appended data may not have the same record structure as the original data, depending on the file's record format.

read Read Data from a File or a Socket

(continued)

```
#include <fcntl.h>
#include <lclib.h>

main()
{
   char fname[80];
   char buffer[80];
   int fd, len;
   /* position of original end of file */
   long endpos;
   /* read and write positions         */
   long rdpos, wtpos;

   puts("Enter the full name of the file to be appended to itself.");
   memcpy(fname, "//dsn:", 6);
   gets(fname+4);
   fd = open(fname, O_RDWR);
   if (fd < 0){
      puts("The file failed to open.");
      exit(EXIT_FAILURE);
   }

   /* Find end of file position. */
   endpos = lseek(fd, 0, SEEK_END);
   rdpos = 0;
   wtpos = endpos;
   for (;;) {
      /* Go to current read position. */
      lseek(fd, rdpos, SEEK_SET);
      /* Read up to 80 bytes.       */
      len = read(fd, buffer, endpos - rdpos > 80?
                 80: endpos - rdpos);

      if (len <= 0){
         puts("Input error - program terminated.");
         exit(EXIT_FAILURE);
      }
      /* Get current position.    */
      rdpos = lseek(fd, 0, SEEK_CUR);
      /* Seek to write position.  */
      lseek(fd, wtpos, SEEK_SET);
      write(fd, buffer, len);
      /* Stop when we've read     */
      /*  to end.                 */
      if (rdpos == endpos) break;
```

read Read Data from a File or a Socket
(continued)

```
            /* Save current position.   */
            wtpos = lseek(fd, 0, SEEK_CUR);
         }
         close(fd);
         exit(EXIT_SUCCESS);
      }
```

RELATED FUNCTIONS

`fread`, **`readv`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

_read Read Data from an HFS File

DESCRIPTION

`_read` is a version of **`read`** designed to operate only on HFS files. **`_read`** runs faster and calls fewer other library routines than **`read`**. Refer to **`read`** for a full description. **`_read`** is used exactly like the standard **`read`** function. The first argument to **`_read`** must be the file descriptor for an open HFS file.

readdir Read Directory Entry

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dir);
```

DESCRIPTION

readdir returns information about the next directory entry from an HFS directory opened by **opendir**. The **dir** function is the value returned by **opendir** when the directory was opened. **readdir** uses a single area for return information for each directory. This means that each call to **readdir** overlays the return information from the previous call for the same directory. Whether or not information is returned for the **``.''** and **``..''** directory entries is not defined by the POSIX.1 standard. Under OpenEdition, these entries are returned.

The **dirent** structure contains the following:

char *d_name	points to a string that names a file in the directory. The string terminates with a null. It has a maximum of NAME_MAX characters.

RETURN VALUE

readdir returns the pointer to a **dirent** structure that describes the next directory entry. **readdir** returns a NULL pointer when it reaches the end of the stream. **readdir** returns a NULL pointer and sets **errno** if it is not successful.

EXAMPLE

The example for **rewinddir** illustrates the use of the **readdir** function.

RELATED FUNCTIONS

opendir

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

readlink Read Symbolic Link

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

int readlink(const char *fn, char *buf, size_t size)
```

DESCRIPTION

readlink reads the contents of a symbolic link. **fn** is the pathname. **buf** is the buffer into which the information is read. **size** is the size of the buffer in bytes.

When you call **readlink** in a non-**posix**-compiled application, the link name is interpreted according to the normal rules for interpretation of file names. The name should include a style prefix if the default style is not **''hfs''**. Also, when **readlink** is called in a non-**posix**-compiled application, the value stored in **buf** has the style prefix **''hfs:''** prepended.

RETURN VALUE

readlink returns the number of bytes in the buffer, or the number of bytes in the symbolic link if the buffer size is stored into 0. **readlink** returns −1 if it is not successful.

EXAMPLE

The example for **symlink** also illustrates the use of **readlink**.

RELATED FUNCTIONS

symlink

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

realloc Change the Size of an Allocated Memory Block

SYNOPSIS

```
#include <stdlib.h>

void *realloc(void *p, size_t size);
```

DESCRIPTION

realloc shrinks or expands a memory block previously allocated by **malloc** or **calloc**, possibly moving it to another location. **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

realloc returns the address of the first character of the new memory block. The reallocated 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 **realloc** returns **NULL**.

ERRORS

User ABEND 1205 or 1206 may occur if memory management data areas are overlaid.

DIAGNOSTICS

If adequate memory is not available or if 0 bytes were requested, **NULL** is returned.

CAUTION

When the reallocated memory block is larger than the original memory block, the contents of the added space are not initialized.

realloc is an inefficient memory allocation tool, especially when used on large blocks. Use linked lists rather than arrays expanded with **realloc** to improve both execution speed and memory use.

IMPLEMENTATION

Under an XA or ESA operating system, memory allocated by **malloc** and reallocated by **realloc** reside above the 16-megabyte line for programs that run in 31-bit addressing mode.

realloc Change the Size of an Allocated Memory Block
(continued)

EXAMPLE

This example doubles the size of a table, if necessary, using **realloc**.

```
#include <stdlib.h>
#include <stdio.h>

char **table, **temp, *item;
unsigned table_size, max_elem;

   /* Determine if table size is too small.              */
if (max_elem >= table_size) {
   table_size *= 2;    /* Double table size.             */

      /* Allocate more space for table. */
   temp = realloc((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);
   }
}
table[max_elem++] = item;
```

RELATED FUNCTIONS

free, **malloc**

SEE ALSO

"Memory Allocation Functions" on page 2-10

remove Delete a File

SYNOPSIS

```
#include <stdio.h>

int remove(const char *name);
```

DESCRIPTION

Under MVS, **remove** deletes the MVS disk file, PDS member, or hierarchical file specified by **name**.

Under CMS, **remove** deletes the CMS disk file specified by the filename that is pointed to by **name**.

RETURN VALUE

The **remove** function returns 0 if the file is deleted. If the file cannot be deleted or **name** is invalid, −1 is returned.

CAUTION

MVS

For **ddn** style filenames that refer to an MVS sequential file, **remove** means to make empty. Other style filenames are deleted and uncataloged.

VSAM data sets cannot be deleted by **remove**.

CMS

If the fileid has a blank filemode, it defaults to **A1**. The string pointed to by **name** should not contain wildcard values such as ***** or **=**.

If the filename is in **xed** style, and XEDIT is not active or the file is not found in XEDIT, the file is searched for on disk. You cannot remove a file found in XEDIT. However, if an **xed** style file is not found in XEDIT but is found on disk, the file is removed. VSAM data sets cannot be deleted by **remove**.

IMPLEMENTATION

Under MVS, **remove** can issue STOW, SCRATCH, CATALOG, OPEN, and CLOSE SVCs. For an HFS file, **remove** calls **unlink**. Under CMS, **remove** performs a CMS FSERASE or invokes the callable service library routine DMSERASE.

remove *(continued)* Delete a File

EXAMPLE

```
/* for MVS systems */

int rc;

   /* Delete a member of the TSO file USELESS.DATA  */
rc = remove("tso:useless.data(removeme)");

   /* Delete a member of the data set pointed to by */
   /*  ddname DATA1 */
rc = remove("ddn:data1(obs11)");

   /* Empty the data set pointed to by ddname DATA1 */
rc = remove("ddn:data1");

/* for CMS systems */

int rc;

rc = remove("cms:oldprog c a");

/* or */

rc = remove("ddn:data1");
```

RELATED FUNCTIONS

`rename`, **`rmdir`**

SEE ALSO

"File Management Functions" on page 2-16

rename

Rename a Disk File

SYNOPSIS

```
include <stdio.h>

int rename(const char *name1, const char *name2);
```

DESCRIPTION

The **rename** function changes the name of an MVS disk file, PDS member, HFS file, CMS disk file, or Shared File System (SFS) file or directory. Under MVS, the character string pointed to by **name1** specifies the filename of an existing MVS disk file or HFS file; **name2** specifies the new MVS filename. Under CMS, the character string pointed to by **name1** specifies the fileid of an existing CMS disk file or SFS file or directory; **name2** specifies the new CMS fileid.

RETURN VALUE

The **rename** function returns 0 if the file is renamed. If the file cannot be renamed or if the name is invalid, −1 is returned.

CAUTION

MVS

The **name1** and **name2** arguments must refer to MVS disk files or HFS files with the same filename style. Nondisk files cannot be renamed. For example, it is impossible to rename a data set residing on tape.

You cannot use the filenames **nullfile**, **sysout=*class***, and **&tmpname** in **rename**. Also, you cannot use **ddn** style filenames that do not also specify a member name when renaming PDS members. The new PDS name must be the same as the old name, and only the member names can differ.

VSAM data sets cannot be renamed by the **rename** function.

CMS

name1 and **name2** must refer to CMS disk files in the **cms** or **xed** style, or CMS Shared File System files or directories using the **sf** or **sfd** style.

The fileids should not contain wild cards such as ***** or **=**. If the filemode in **name2** is blank, it is replaced with the filemode of **name1**. If the filemode in **name1** is blank, it is replaced by the default mode **A1**. If **name2** refers to an existing fileid, a message is printed and −1 is returned. When you use the CMS Shared File System, you cannot use **rename** to change a file's directory or file pool.

IMPLEMENTATION

Under MVS, **rename** can issue STOW, CATALOG, and RENAME SVCs. Under CMS, **rename** issues a CMS RENAME command or the CMS callable services library routine DMSRENAM.

EXAMPLE

```
#include <stdlib.h>
#include <stdio.h>

main()
```

rename Rename a Disk File

(continued)

```
{
   int rc;

      /* Rename a dataset TESTFILE to EXAMPLE. */
   rc = rename("tso:testfile",
               "tso:example");

   if (rc == 0)
     puts("Dataset has been renamed.");
   else
     puts("Dataset not renamed.");
}
```

RELATED FUNCTIONS

`remove`

SEE ALSO

"File Management Functions" on page 2-16

_rename Rename an HFS File

DESCRIPTION

`_rename` is a version of **`rename`** designed to operate only on HFS files. **`_rename`** runs faster and calls fewer other library routines than **`rename`**. Refer to **`rename`** for a full description. **`_rename`** is used exactly like the standard **`rename`** function. The arguments to **`_rename`** are interpreted as HFS filenames, even if they appear to begin with a style prefix or a leading `//` or both.

rewind Position to Start of File

SYNOPSIS

```
#include <stdio.h>

void rewind(FILE *f);
```

DESCRIPTION

`rewind` positions the stream associated with the **`FILE`** object addressed by **`f`** to its first character. It also resets the error flag for the stream if it is set.

RETURN VALUE

`rewind` has no return value.

EXAMPLE

This example searches for the *n*th record in a file, returns that record, and rewinds the file after the search is finished:

```
#include <stdio.h>
#define RECLEN 80

FILE *f;           /* file to be searched                  */
char *search(int); /* prototype of the search function     */

main()
{
   int n ;         /* The record number to be found        */
                   /*  points to the address of a copy of */
                   /*  the returned record.                */
   char *addr;

      /* Ask the user for the number of the record to      */
      /*  be found.                                        */
   puts("Which record do you want to read?");
   scanf("%d", &n);
   addr = search(n);
   printf("The record is %s\n", *addr);
}
   /* performs the search and rewind of the file f         */
char *search(int n){
   char *record;   /* points to a copy of the record       */
   int i;
   while (!feof(f)) {

         /* Read the records until the nth one is found.  */
      for (i=0; i <= n; i++)
         afread(record, RECLEN, 1, f);
   }

      /* Reposition the stream to the top of f.            */
   rewind(f);
```

rewind Position to Start of File
(continued)

```
      /* Return the address of the copy of the record.    */
   return record;
}
```

RELATED FUNCTIONS

`fseek`, **`fsetpos`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1
“I/O Functions” on page 2-13

rewinddir Rewind Directory Stream

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <dirent.h>

void rewinddir(DIR *dir);
```

DESCRIPTION

rewinddir positions a directory stream to the beginning. **dir** is a pointer to an object associated with the open directory by a call to **opendir**. After a call to **rewinddir**, the next call to **readdir** reads the first entry in the directory. If the contents of the directory have changed since the directory was opened, **rewinddir** updates the directory stream for the next call to **readdir**.

RETURN VALUE

rewinddir returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

```
#include <sys/types.h>
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>

main()
{
   DIR *dir;
   struct dirent *curfile;
   int fd;
   char flname[] = "testdir";

   if ((dir=opendir(".")) == NULL)
      perror("opendir() error");
   else {
      puts("Root directory contents:");
      while ((curfile = readdir(dir)) != NULL)
            /* Print the current directory contents. */
         printf("\n%s ", curfile->d_name);
         /* Create test directory.                 */
      if ((fd=mkdir(flname,S_IWUSR)) < 0)
         perror("mkdir() error");
      rewinddir(dir);
      puts("\nCurrent directory contents");
      while ((curfile = readdir(dir)) != NULL)
         printf("\n%s ", curfile->d_name);
         /* Remove testdir directory.              */
```

rewinddir Rewind Directory Stream

(continued)

```
         if (rmdir(flname) != 0)
            perror("rmdir error");
         else
            printf("\nThe testdir directory %s has been removed.\n",
            flname);
         }
      closedir(dir);
   }
```

RELATED FUNCTIONS

`opendir`

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

rmdir Remove Directory

POSIX UNIX

SYNOPSIS

```
#include <unistd.h>

int rmdir(char *pathname);
```

DESCRIPTION

`rmdir` removes an empty directory named by **`pathname`**. The **`rmdir`** function deletes the directory itself if no process has the directory open, and the space that was occupied by the directory is freed. **`pathname`** is removed even if it is the working directory of a process. If a process has the directory open, **`unlink`** removes the links, but the directory is not removed until the last process has closed it. You cannot create new files in the directory after the last link is removed.

RETURN VALUE

`rmdir` returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

See the example for **`rewinddir`**.

RELATED FUNCTIONS

`mkdir`

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

“File Management Functions” on page 2-16

sbrk UNIX Low-Level Memory Allocation

SYNOPSIS

```
#include <lclib.h>
char *sbrk(size_t bytes);
```

DESCRIPTION

sbrk allocates a block of memory of the size specified by **bytes**. The block is suballocated from an area allocated at program initialization. The size of this area is determined by the initial value of the external variable **_mneed**; if this variable is not set, a default area of 100K is allocated the first time **sbrk** is called.

RETURN VALUE

sbrk returns the address of the first character of the block of memory. The block is suitably aligned for storage of any type of data.

CAUTION

The contents of a memory block on allocation are random.

sbrk is an inflexible mechanism for memory allocation. It has no provision for increasing the size of the **sbrk**-managed area (even if free memory is available for this purpose). Using **malloc**, which does not have this limitation, is recommended for memory allocation wherever possible.

Memory allocated with **sbrk** cannot be returned to the operating system (except implicitly at program termination).

DIAGNOSTICS

If adequate memory is not available when **sbrk** is called or if 0 bytes of memory are requested, **sbrk** returns **(char *)-1**.

PORTABILITY

sbrk is compatible with some versions of traditional UNIX C compilers. It is not well suited to the 370 environment; therefore, use **malloc** in its place whenever possible.

IMPLEMENTATION

Under an XA or ESA operating system, memory allocated by **sbrk** resides above the 16-megabyte line for programs that run in 31-bit addressing mode.

sbrk UNIX Low-Level Memory Allocation
(continued)

EXAMPLE

```
#include <lclib.h>
#include <stdio.h>

int _mneed = 1024;       /* Define default size of sbrk area. */

main()
{
   int n;
   char *stg;

   for(n = 1; ; ++n){
      stg = sbrk(80);
      if (stg == (char *) -1) break;
   }
   printf("%d 80-byte blocks could be allocated by sbrk.\n", n);
   puts("To change the amount available to sbrk, pass "
         "the runtime option =/<n>,");
   puts("replacing <n> with the size of the sbrk area.");
}
```

RELATED FUNCTIONS

malloc

SEE ALSO

"Memory Allocation Functions" on page 2-10

scanf Read Formatted Data from the Standard Input Stream

SYNOPSIS

```
#include <stdio.h>

int scanf(const char *format, loc1, ... );
```

DESCRIPTION

`scanf` reads formatted data from **`stdin`**. Following the format in the argument list may be one or more additional pointers (**`loc1, loc2, ..., locn`**) addressing storage where the input values will be stored.

The string pointed to by **`format`** is in the same form as that used by **`fscanf`**. Refer to the **`fscanf`** description for detailed information concerning the formatting conventions.

RETURN VALUE

`scanf` returns **`EOF`** if end of file (or an input error) occurs before any values are stored. If any values are stored, it returns the number of items stored; that is, it returns the number of times a value is assigned by one of the **`scanf`** argument pointers.

DIAGNOSTICS

`EOF` is returned if an error occurs before any items are matched.

IMPLEMENTATION

`scanf` is identical to **`fscanf`** with **`stdin`** as the input file.

EXAMPLE

```
#include <lcio.h>
#include <stdio.h>

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 (-1 to indicate end of list).");
```

scanf Read Formatted Data from the Standard Input Stream
(continued)

```
      for(;;){
            /* Read number; check for end of file. */
         if (scanf("%le", &point[index]) <= 0)
            break;
         if (point[index] == -1)
         break;
         sum += point[index];
         ++index;
      }
      nopoints = index;
      avg = sum / nopoints;
      printf("%d points read.\n", nopoints);
      printf("%f = average.\n", avg);
   }
```

RELATED FUNCTIONS

`fscanf`, **`sscanf`**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

setbuf Change Stream Buffering

SYNOPSIS

```
#include <stdio.h>

void setbuf(FILE *stream, char *buf);
```

DESCRIPTION

`setbuf` controls buffering for the specified **`stream`** on operating systems that support user-defined buffers. **`setbuf`** is similar to the **`setvbuf`** function. If **`buf`** is not **`NULL`**, then the values _IOFBF and BUFSIZE are used for **`setvbuf`**'s **`mode`** and **`size`** arguments. If **`buf`** is **`NULL`**, then the value _IONBF is used for **`setvbuf`**'s **`mode`** argument.

For FILE pointers that reference HFS files or sockets, you can use **`setbuf`** to change the buffering mode or location. If you do not use **`setbuf`**, the default buffer size for HFS files and sockets is 1008 bytes. **`setbuf`** has no effect for any other kind of file. A call to **`setbuf`** is permitted only as the first operation following the opening of a file.

RETURN VALUE

`setbuf` has no return value.

EXAMPLE

```
#include <stdio.h>

main()
{
   char input[32];

   setbuf(stdout, NULL); /* Try to prevent buffering of stdout. */
   printf("Please enter your first name:");
   fflush(stdout);       /* Try to force output to terminal.    */
   gets(input);
   printf("Thanks %s. It's been a pleasure.\n",input);
}
```

RELATED FUNCTIONS

`setvbuf`

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

setenv Assign Environment Variable

SYNOPSIS

```
#include <lclib.h>
int setenv(const char *name, const char *value);
```

The synopsis for the POSIX implementation is

```
#include <stdlib.h>
int setenv(const char *name, const char *value)
```

You should use **`<stdlib.h>`** only if an appropriate feature test macro has been defined.

DESCRIPTION

`setenv` creates an environment variable with a specified name and value, or assigns a new value to an existing environment variable. **`name`** and **`value`** are specified by the string pointed to by the argument string.

The format of **`name`** is

[[*scope*]:][[*groupname*].]*varname*

For portable use, the *scope* and *groupname* parts of the string must be omitted. The parts are

scope
: specifies the scope at which the environment variable is to be added or modified. It may be specified as

  PRogram

  External (or STorage)

  PErmanent (or Lasting)

  SEssion.

  scope is not case sensitive. The uppercase letters indicate the minimum abbreviation that may be specified for the *scope* name. See Chapter 4, “Environment Variables” on page 4-1 for a definition of the environment-variable scopes. The SEssion scope is CMS specific and refers to GLOBALV SESSION variables. For all other systems, a SEssion-scope specification is treated as if it were an External-scope specification. If you do not specify scope, PRogram scope is assumed. Scopes other than PRogram are valid only under TSO, CMS, and CICS.

groupname
: specifies an optional group name for the environment variable. *groupname* is only meaningful for nonprogram-scope variables. If a program-scope, environment-variable name appears to have a group name, the group name is simply treated as part of the variable name. See Chapter 4, “Environment Variables” on page 4-1 for more information.

varname
: specifies the name of the environment variable. For nonprogram-scope variables, some environments restrict the size of the variable name. In these environments, the name is truncated, if necessary.

setenv Assign Environment Variable
(continued)

RETURN VALUE

setenv returns 0 if it is successful, or −1 if it is unsuccessful.

CAUTION

Do not modify the environment by changing the external variable **environ** or the data it points to in a program that uses **setenv**. The **setenv** function may cause the value of **environ** to change.

PORTABILITY

Environment variable scopes and groups are SAS/C extensions and should not be used in portable programs.

USAGE NOTES

The same variable name can be set in each scope. However, **setenv** always returns the value of shortest duration. For example, if a program **scope** variable is defined, **setenv** always returns its value.

EXAMPLE

This example creates an environment variable named **HOME**, if it does not already exist, and then invokes an OpenEdition shell command:

```
#include <stdio.h>
#include <lclib.h>
#include <lcstring.h>

main() {
   char home[12];
   char cmd[300];
   int rc;

         /* if environment variable HOME not defined           */
      if (!getenv("HOME")) {
      strcpy(home, "/u/");
      cuserid(home+3);           /* Append userid to directory */
                                 /*  name.                     */
      strlwr(home);              /* Translate to lowercase     */
                                 /*  letters.                  */
      rc = setenv("HOME", home); /* Define HOME.               */
      if (rc != 0) {
         perror("setenv failure");
         exit(EXIT_FAILURE);
      }
   }
   puts("Enter shell command");
   memcpy(cmd, "//sh:", 5);      /* prefix for system function */
   gets(cmd+5);
   rc = system(cmd);             /* Invoke the shell command.  */
   printf("shell command status code was %d.\n", rc);
   exit(rc);
}
```

setenv Assign Environment Variable
(continued)

RELATED FUNCTIONS

`clearenv`, **`getenv`**, **`putenv`**

SEE ALSO

Chapter 6, “Executing C Programs,” in *SAS/C Compiler and Library User’s Guide*

Chapter 4, “Environment Variables” on page 4-1

“System Interface and Environment Variables” on page 2-17

setjmp Define Label for Nonlocal goto

SYNOPSIS

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

DESCRIPTION

setjmp defines a target for a nonlocal **goto**. The call to **setjmp** always returns 0. If another routine, called later by the caller of **setjmp**, issues the call **longjmp(env, code)**, the earlier call to **setjmp** is resumed. This time, **setjmp** returns the value contained in the **code** argument to **longjmp**.

RETURN VALUE

A true return from **setjmp** always produces a 0. When control returns from **setjmp** because **longjmp** was used, the return value is nonzero.

CAUTION

Variables of storage class **auto** and **register**, whose values have been changed between the **setjmp** and **longjmp** calls, have indeterminate values on return to **setjmp** unless declared volatile.

EXAMPLE

```
#include <stdio.h>
#include <setjmp.h>
#include <stdlib.h>

jmp_buf env;
void dummy();

main()
{
   int ret;

   if ((ret = setjmp(env)) != 0) {
      fprintf(stderr, "longjmp called with value %d\n", ret);
      exit(1);
   }
   dummy();
   fprintf(stderr, "longjmp was not called.\n");
}

void dummy()
{
   puts("Entering dummy routine.");
   longjmp(env, 3);
   puts("Never reached.");
}
```

setjmp Define Label for Nonlocal goto
(continued)

RELATED FUNCTIONS

`longjmp`, **`sigsetjmp`**

SEE ALSO

"Program Control Functions" on page 2-9

setvbuf Change Stream Buffering

SYNOPSIS

```
#include <stdio.h>

int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

DESCRIPTION

setvbuf requests a buffering mode for a stream. **stream** is a pointer to an open file on which no other operation has been performed. **buf** specifies the area that the C library uses as the buffer for **stream**. The **mode** function can have one of three values, defined as macros in **<stdio.h>**:

_IOFBF	indicates full buffering.
_IOLBF	indicates line buffering.
_IONBF	indicates no buffering.

size must be greater than zero. If **buf** is not **NULL**, then the array it points to may be used instead of a buffer allocated by **setvbuf**. For **buf**, the length in bytes is indicated by **size**.

For FILE pointers that reference HFS files or sockets, you can use **setvbuf** to change the buffering mode, or the buffer size or location, or both. If **setvbuf** is not used, the default buffer size for HFS files and sockets is 1008 bytes. For all other file types, **setvbuf** has no effect. **setvbuf** is permitted only as the first operation following the opening of a file.

RETURN VALUE

setvbuf returns 0 if the stream can be buffered as specified by **mode**, or nonzero if the request cannot be honored.

IMPLEMENTATION

In this implementation, the buffering mode of a non OpenEdition stream is chosen when the file is opened and cannot be changed by **setvbuf**. For non OpenEdition streams, **setvbuf** returns 0 if the value of **mode** is the same as the buffering mode chosen when the file is opened. If the file is opened as a text stream, then a **mode** value of _IOLBF causes **setvbuf** to return 0. If the file is opened as a binary stream, then a **mode** value of _IOFBF causes **setvbuf** to return 0.

EXAMPLE

This example counts the number of characters in a text file. It uses the **setvbuf** function to request a 4K buffer for reading the file. On some systems, this may improve speed of access; on MVS or CMS, this has no effect.

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 4000

char *_style = "tso"; /* Use TSO-style filenames by default. */
```

setvbuf Change Stream Buffering

(continued)

```
main()
{
  FILE *in;
  int ch;
  char fname[80];
  int count = 0;

  puts("Enter the file you want to read:");
  gets(fname);
  in = fopen(fname, "r");
  if (!in){
     puts("That file cannot be opened.");
     exit(EXIT_FAILURE);
  }
  setvbuf(in, NULL, _IOFBF, BUFFER_SIZE);
     /* Ask for a large buffer.                              */
  while((ch = getc(in)) != EOF) ++count;
  printf("That file contains %d characters.\n", count);
  exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

`setbuf`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

sfsstat Return Information about a CMS Shared File System File or Directory

SYNOPSIS

```
#include <cmsstat.h>

int sfsstat(const char *path, struct sfsstat *buf);
```

DESCRIPTION

The **sfsstat** function fills in an **sfsstat** structure with system-dependent information about a Shared File System (SFS) file or directory. You can specify the **path** parameter as any name in the **sf** or **sfd** style. The **buf** parameter must point to an **sfsstat** structure as defined in **<cmsstat.h>**. The **sfsstat** structure is defined as

```
struct sfsstat {
  time_t        st_updt;     /* date and time last updated     */
  time_t        st_crdt;     /* date and time file was created */
  unsigned int  st_type;     /* device type flag               */
                             /*  S_DISK - disk file            */
                             /*  S_SFS -shared file system file*/
                             /*  S_SFSDIR - sfs directory      */
                             /*  S_DIRCNTL -sfs dircontrol dir */
                             /*  S_FILCNTL -sfs filecontrol dir*/
  char  st_owner[9];         /* null-terminated owner id       */
  char  st_dir[154]          /* null-terminated directory name */
  char  st_fname[9];         /* null-terminated CMS filename   */
  char  st_ftype[9];         /* null-terminated CMS filetype   */
  char  st_fmno[2];          /* null-terminated CMS fm number  */
  char  st_flags;            /* access flags                   */
                             /*  S_RW  - read/write            */
                             /*  S_RO  - only readable         */
                             /*  S_NO  - not read or write     */
                             /*  S_EP  - externally protected  */
  int           st_dirlen;   /* length of directory name       */
  int           st_numblks;  /* number of blocks in file       */
  unsigned int  st_lrecl;    /* logical record length (LRECL)  */
  int           st_norecs;   /* number of logical records      */
  char          st_recfm;    /* record format (RECFM)          */
  char          st_status;   /* '1'= base, '2'= alias,         */
                             /*  '3'= erased, '4'=revoked      */
  char          st_dirauth;  /* special directory authorities  */
                             /*  S_NR  - NewRead auth on dir   */
                             /*  S_NW  - NewWrite auth on dir  */
                             /*  S_AR  - Accessed Read only    */
                             /*  S_AW  - Accessed Write        */
  char          st_resrv1;   /* reserved                       */
  unsigned int  st_resrv2;   /* reserved                       */
  unsigned int  st_resrv3;   /* reserved                       */
  unsigned int  st_resrv4;   /* reserved                       */
  unsigned int  st_resrv5;   /* reserved                       */
  };
```

sfsstat Return Information about a CMS Shared File System File or Directory
(continued)

The `st_type` flag can have one of, or a combination of, these values:

S_DISK	indicates a CMS disk file.
S_SFS	indicates an SFS file.
S_SFSDIR	indicates an SFS directory.
S_DIRCNTL	indicates an SFS directory-control directory.
S_FILCNTL	indicates an SFS file-control directory.

The `st_flags` access flag can have one of these values:

S_RW	indicates read or write authority to a file or directory.
S_RO	indicates only read authority to a file or directory.
S_NO	indicates no authority to a file or directory.
S_EP	indicates a file or directory is externally protected.

The `st_status` flag can have one of these values:

1	indicates the file is a base file.
2	indicates the file is an alias.
3	indicates the file is erased.
4	indicates the authority for the file is revoked.

The `st_dirauth` flag can have one of, or a combination of, these values:

S_NR	indicates new-read authority to the directory.
S_NW	indicates new-write authority to the directory.
S_AR	indicates the directory is accessed as a minidisk in read-only status.
S_AW	indicates the directory is accessed as a minidisk in read or write status.

RETURN VALUE

If the file exists and the program has at least read authority to the parent directory and the file, `sfsstat` returns 0 and fills in the appropriate fields in the `sfsstat` structure. If the file does not exist, the program has insufficient authority, or the fileid is invalid, `sfsstat` returns −1.

CAUTION

Some fields in the `sfsstat` structure may not be useful for all successful calls to `sfsstat`. For some files, some of the fields of the `sfsstat` structure are not applicable or are unavailable in some releases of CMS.

sfsstat Return Information about a CMS Shared File System File or Directory
(continued)

The values returned for each such field are

Function	**Return Values**
`sfsstat.st_updt`	`(time_t)` − 1
`sfsstat.st_crdt`	`(time_t)` − 1
`sfsstat.st_fname`	“”
`sfsstat.st_ftype`	“”
`sfsstat.st_fmno`	“”
`sfsstat.st_recfm`	`0xffff`
`sfsstat.st_lrecl`	`0xffff`
`sfsstat.st_numblks`	− 1
`sfsstat.st_norecs`	− 1
`sfsstat.st_status`	`0xffff`
`st_dirauth`	’\0’

Fields in the **`sfsstat`** structure may have been modified, even if the function returns −1.

IMPLEMENTATION

The CMS callable services library routine DMSEXIST is invoked.

EXAMPLE

```
#include <cmsstat.h>
#include <stdio.h>

main()
{
   struct sfsstat info;
   int rc;
   rc = sfsstat("sf:my file cuser.subdir1", &info);
   if (rc == 0){
      puts("access field (auth)");
      if (info.st_flags == S_RW)
         puts("   st_flags   = S_RW");
      if ((info.st_flags & S_RO) ==      S_RO)
         puts("   st_flags   = S_RO");
      if ((info.st_flags & S_NO) ==      S_NO)
         puts("   st_flags   = S_NO");
```

sfsstat Return Information about a CMS Shared File System File or Directory
(continued)

```
         if ((info.st_flags & S_EP) ==      S_EP)
            puts("   st_flags   = S_EP");
      }
      return rc;
   }
```

RELATED FUNCTIONS

`cmsstat`

SEE ALSO

"File Management Functions" on page 2-16

sigaction Define a Signal-Handling Action

SYNOPSIS

```
#include <lcsignal.h>

int sigaction(int signum, const struct sigaction *newsig,
              struct sigaction *oldsig);
```

The synopsis for the POSIX implementation is

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *newsig,
              struct sigaction *oldsig);
```

You should use **<signal.h>** only if an appropriate feature test macro has been defined.

DESCRIPTION

sigaction modifies the action associated with a signal. **sigaction** can be used for signals defined by SAS/C as well as OpenEdition signals. It does not require that OpenEdition be installed or available. **signum** is the number of the signal. **signum** must be a symbolic signal name defined in **<signal.h>**. **newsig** is the new action associated with a signal; if **newsig** is a NULL pointer, the signal action is not changed. **oldsig** is a pointer to a location where the action currently associated with the signal is to be stored, or NULL if this information is not needed.

The **sigaction** structure is defined as

```
struct sigaction {
      /* function that handles the signal          */
   __remote void (*sa_handler)(int);
      /* set of signals to be added to signal mask */
   sigset_t sa_mask;
      /* signal flags                              */
   int sa_flags;
};
```

sa_handler can be a pointer to a function, or it can have one of these values:

SIG_DFL	is the default
SIG_IGN	specifies that the signal should be ignored.

sa_mask is the set of signals to be added to the signal mask during handling of the signal.

sa_flags enables these flag bits (as defined by the POSIX.2 standard) to be set:

SA_NOCLDSTOP	prevents a SIGCHLD signal from being issued when a child process terminates.

sigaction Define a Signal-Handling Action
(continued)

SAS/C also defines a number of nonstandard flags that may be set in the **`sa_flags`** word of the **`sigaction`** structure. The relevant nonstandard flags are

`SA_GLOBAL`
: specifies that the signal handler is defined as a global signal handler; that is, one that applies to all coprocesses. (A **`sigaction`** call with **`SA_GLOBAL`** set is comparable in effect to a **`cosignal`** call.)

  Note: In an application with coprocesses, the **`sa_mask`** and **`sa_flags`** settings for a call to **`sigaction`** always apply to all coprocesses, not just to the calling coprocess. See Chapter 9, "Coprocessing Functions," in *SAS/C Library Reference, Volume 2* for more details.

`SA_PREVIOUS`
: specifies that the **`sa_mask`** and **`sa_flags`** values specified by the argument to sigaction are to be ignored, and the settings specified by the previous call to **`sigaction`** are to be used. This flag is useful for defining local handlers in a coprocess without perturbing the handling defined by other coprocesses. See Chapter 9, "Coprocessing Functions," in *SAS/C Library Reference, Volume 2* for more details.

`SA_USRFLAG1`, **`SA_USRFLAG2`**, ... **`SA_USRFLAG8`**
: specify options to user-defined signal handlers. Their meaning, if any, is defined by the signal implementor. These flags have no meaning for any signal defined by SAS/C.

RETURN VALUE

`sigaction` returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

See the example for **`sigsetjmp`**.

RELATED FUNCTIONS

`cosignal`, **`sigaddset`**, **`signal`**, **`sigprocmask`**

SEE ALSO

Chapter 9, "Coprocessing Functions," in *SAS/C Library Reference, Volume 2*

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

sigaddset, sigdelset, sigemptyset, sigfillset, sigismember

Modify the Signals in a Set of Signals

SYNOPSIS

```
#include <lcsignal.h>

int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigismember(const sigset_t *set, int signum);
```

For the POSIX implementation, include the header file **`<signal.h>`** after defining an appropriate feature test macro.

DESCRIPTION

`sigaddset`, **`sigdelset`**, **`sigemptyset`**, **`sigfillset`**, and **`sigismember`** enable you to modify the **`sigset_t`** object addressed by **`set`**. The **`sigset_t`** function is defined in **`<signal.h>`** and **`<lcsignal.h>`** as a type representing a collection of signal numbers. Signal sets are passed as arguments to other signal handling functions, such as **`sigprocmask`**, **`sigpending`**, and **`oesigsetup`**.

The functions that modify signals in a set of signals include

`sigaddset`	adds the signal **`signum`** to the set.
`sigdelset`	deletes the signal **`signum`** from the set.
`sigemptyset`	initializes a set to contain no signals.
`sigfillset`	initializes a set to contain all signals.
`sigismember`	tests to see if **`signum`** is a member of the set.

RETURN VALUE

`sigismember` returns 1 if **`signum`** is a member of the set, or 0 if it is not in the set. All other functions return 0 if successful. All functions return −1 if an error occurs (such as an invalid signal number).

EXAMPLE

The following example uses these functions to set up signal sets for **`oesigsetup`**. The call to **`oesigsetup`** defines **`SIGALRM`** and **`SIGFPE`** as signals managed by SAS/C, and all others as signals managed by OpenEdition. See the **`sigpending`** example for an example using **`sigismember`**.

```
#include <lcsignal.h>

sigset_t sascset, oeset;
sigemptyset(&sascset);
sigaddset(&sascset, SIGFPE);
/* SAS/C will manage SIGALRM and SIGFPE. */
sigaddset(&sascser, SIGALRM);
sigfillset(&oeset);
sigdelset(&oeset, SIGFPE);
```

sigaddset, sigdelset, sigemptyset, sigfillset, sigismember
(continued)

Modify the Signals in a Set of Signals

```
   /* OpenEdition manages everything else   */
   /* (whenever possible).                  */
   sigdelset(&oeset, SIGALRM);

   oesigsetup(&oeset, &sascset);
```

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

sigblock Inhibit Discovery of Asynchronous Signals

SYNOPSIS

```
#include <lcsignal.h>

int sigblock(int mask);
```

DESCRIPTION

sigblock delays discovery of one or more asynchronous signals. The **mask** argument is an integer that is interpreted as a bit string. You use this bit string to alter a mask of blocked signals. For example, the following call requests that the SIGALRM signal be blocked:

```
sigblock(1<<(SIGALRM - 1));
```

You can use this format to block any single asynchronous signal managed by SAS/C; simply change the name of the signal to be blocked.

By specifying a mask of 0, you also can use **sigblock** to determine what signals are currently blocked without making any changes.

The SAS/C library honors only bits corresponding to the asynchronous signals (SIGINT, SIGALRM, SIGIUCV, and SIGASY1 through SIGASY8); any other bits set in the mask are ignored. **sigblock** changes the status of only the bits specified in the argument. All other bits in the mask are unchanged; that is, if any of them were previously blocked, they remain blocked. Also, **sigblock** does not affect any signals managed by OpenEdition. For this reason, **sigprocmask**, which can be used for all signals, is preferable to **sigblock**.

If a signal occurs while it is blocked, the signal is kept pending until the signal is unblocked by a call to **sigsetmask**, **sigpause**, **sigprocmask**, **sigsuspend**, **ecbpause**, or **ecbsuspend**. When the signal is unblocked, it is discovered, and the appropriate handler is called. Refer to "Blocking Signals" on page 5-14 for more information.

For compatibility with existing programs, a call to **sigblock** requesting that all signals be blocked (a signal mask of all ones) causes all blockable OpenEdition signals to be blocked as well. This blocking occurs within the library, so if you call **sigblock(0xffffffff)** and then use an **exec** function to transfer control to another program, that program receives control with no signals blocked.

RETURN VALUE

sigblock returns the previous mask of blocked signals. You can pass this value to the **sigsetmask** function to restore the previous set of blocked signals. Bits of the mask corresponding to synchronous signals are always 0.

CAUTION

You should not keep signals blocked for long periods of time because this may use large amounts of memory to queue pending signals. For lengthy programs, you should use **sigblock** to protect critical sections of the program and then reset the mask with **sigsetmask** to enable signals to occur freely in less critical areas.

sigblock Inhibit Discovery of Asynchronous Signals
(continued)

The library sometimes uses **sigblock** to delay asynchronous signals during its own processing. If the library is in the middle of processing and something occurs that causes it to call **longjmp** to return to your program, the mask set by the library may still be in effect; that is, the mask may not be what you specified in your program. For example, suppose a library function runs out of stack space and raises SIGMEM, and the handler for SIGMEM returns to your program with a **longjmp**. You may need to issue **sigsetmask** at the completion of the jump to restore the signal mask needed by the program. The functions **sigsetjmp** and **siglongjmp** may be useful in these situations.

A signal generated by the program calling **raise** or **siggen** always occurs immediately, even if the signal is blocked.

PORTABILITY

sigblock is not portable, except to BSD-compatible UNIX systems.

EXAMPLE

The following code fragment illustrates the use of **sigblock**.

```
#include <lcsignal.h>

int old_mask;

   /* Hold up terminal attentions.    */
old_mask = sigblock(1<<(SIGINT-1));

   /* Perform critical code.          */
.
.
.
   /* Allow same interruptions again. */
sigsetmask(old_mask);
```

RELATED FUNCTIONS

sigpause, **sigprocmask**, **sigsetmask**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

sigchk Check for Asynchronous Signals

SYNOPSIS

```
#include <lcsignal.h>

void sigchk(void);
```

DESCRIPTION

sigchk is called to check for the occurrence of asynchronous signals. If a signal is pending and not blocked, it is handled by the handler defined by your program or by the default handler if none is defined. If more than one signal is pending and not blocked, they are processed in the order in which they occurred. However, for signals managed by OpenEdition, the order in which they are discovered is determined by OpenEdition. In this case, the first signal processed is not necessarily the one that occurred first.

Since asynchronous signals are discovered only when a function is called or returns, **sigchk** is useful for discovering these signals in portions of code that do not call functions (for example, in the middle of a lengthy calculation).

You can use the **sigchk** function to check for the occurrence of any signal, whether managed by SAS/C or by OpenEdition. When a SAS/C handler is defined for a signal, the timing of signal discovery is the same for signals generated by OpenEdition as it is for signals generated by SAS/C. That is, the signal is discovered only during a function call, a function return, or a call to **sigchk**. However, if default handling is specified for an OpenEdition signal, this action occurs as soon as the signal is generated.

RETURN VALUE

No information is returned by **sigchk**. It is not possible to determine directly whether any signals were handled.

IMPLEMENTATION

sigchk is a built-in function; that is, it is implemented by compiler-generated code rather than as a true subroutine call. If no signals are pending when **sigchk** is called, only two instructions are executed.

EXAMPLE

This code fragment uses **sigchk** to perform iterative computations and checks for signals every 100 iterations:

```
#include <lcsignal.h>

for (i = 0; i < 10000; ) {
   for (j = 0; j < 100; ++j,++i) {

      /* Perform calculations. */
   .
   .
   .
   }
sigchk();
}
```

sigchk Check for Asynchronous Signals
(continued)

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1
"Signal-Handling Functions" on page 2-17

siggen Generate an Artificial Signal with Additional Information

SYNOPSIS

```
#include <lcsignal.h>

int siggen(int signum, void *info);
```

DESCRIPTION

siggen artificially generates a signal and provides information about the signal. The argument **signum** is the number of the signal that is to be generated. You should specify this argument symbolically using one of the signal names listed in “Supported Signals” on page 5-3 . When you call **siggen**, the signal is handled as established by a previous call to **signal** or by the default handler if **signal** is not called.

The argument **info** defines the value to be returned by the **siginfo** function, if **siginfo** is called by the signal handler. For some signals, such as SIGABRT, the **info** pointer is also meaningful to the default handler for the signal. If default handling is in effect for an OpenEdition signal generated by **siggen**, **siggen** invokes the **kill** function to enable OpenEdition to perform default handling.

A signal generated by **siggen** is always handled immediately, regardless of whether the signal is asynchronous or whether it is blocked.

RETURN VALUE

siggen returns 0 if it is successful or a nonzero value if it is not successful. Note that **siggen** may not return if the handler for the signal terminates with a call to **longjmp**, **exit**, or **abort**, or if the handler for the signal defaults to abnormal program termination.

CAUTION

If **info** is not **NULL**, it should be a value of the appropriate pointer type.

See the description of the signal for the information returned by a call to **siginfo** when a signal occurs naturally. For example, the expected type for the SIGSEGV signal is **SEGV_t.** Declarations for these types (except the types for SIGIUCV) are included in **<lcsignal.h>**.

If you use **siggen** to test signal handling in a program, keep in mind that signals generated by **siggen** are treated somewhat differently than signals that occur naturally. Signals generated by **siggen** are always handled immediately, so you should not use **siggen** to test signal blocking. You may also encounter problems if you use **siggen** to test the handler for an asynchronous signal because the handler for the signal is executed immediately, not just when a function is called or returns.

EXAMPLE

The following code fragment illustrates the use of **siggen**.

```
#include <lcsignal.h>

/* Abort execution with ABEND code 611. */
```

siggen Generate an Artificial Signal with Additional Information
(continued)

```
ABRT_t abrt_info;
abtr_info.ABEND_str = "";
abrt_info.ABEND_info = NULL;
abrt_info.ABEND_code = 611;
.
.
.
siggen(SIGABRT, &abrt_info);
```

RELATED FUNCTIONS

`abort`, **`kill`**, **`raise`**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1
“Signal-Handling Functions” on page 2-17

siginfo Obtain Information about a Signal

SYNOPSIS

```
#include <lcsignal.h>

void *siginfo(void);
```

DESCRIPTION

siginfo returns information about a signal that is being handled. The value returned by **siginfo** is of type **void ***; it generally must be converted to some other signal-specific type to use the information. For signals that occur naturally, the data addressed by the **siginfo** pointer are signal dependent but provide information about the causes of the interrupt. With some signals, such as SIGFPE, the pointer returned by **siginfo** also addresses data that can be modified to change the value of an erroneous expression. Refer to the description of each signal for details on what is returned by a call to **siginfo** when a signal occurs naturally. In addition, Table 5.1 summarizes what **siginfo** returns for each signal.

If a signal is generated artificially by a call to **siggen**, the value returned by **siginfo** is the same as the second argument to **siggen**. If a signal is generated artificially by a call to **raise**, the value returned by **siginfo** is **NULL**.

If more than one signal handler is active at the time **siginfo** is called, information is returned for the signal whose handler was called most recently.

RETURN VALUE

When **siginfo** is called in a signal handler, it returns a pointer to information associated with the signal being handled. The pointer may need to be converted to some other type before using. If **siginfo** is called outside a handler, it returns **NULL**.

The return value of **siginfo** for any signal managed by OpenEdition is always **NULL**, unless the signal was generated by using the **siggen** function, or the signal was a program check or ABEND directly associated with a program error, such as a SIGFPE signal caused by program division by zero.

EXAMPLE

See the example for **signal**.

RELATED FUNCTIONS

siggen

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

siglongjmp Restore Stack Environment and Signal Mask

SYNOPSIS

```
#include <lcjmp.h>

void siglongjmp(sigjmp_buf stack, int value)
```

The synopsis for the POSIX implementation is

```
#include <setjmp.h>

void siglongjmp(sigjmp_buf stack, int value)
```

DESCRIPTION

siglongjmp restores a previously saved stack environment and signal mask. **stack** is the address for the **sigjmp_buf** structure that contains the stack environment. The stack environment and signal mask must have been saved by **sigsetjmp**. The mask is restored only if it was saved by **sigsetjmp**. Control is returned to the point in the program corresponding to the call to **sigsetjmp**. Execution continues from this point, just as if the **sigsetjmp** function had returned the value specified by the **value** argument.

The **siglongjmp** function restores the old signal mask before jumping to the location defined in the **sigjmp_buf** structure. Because of this, if the jump is intercepted by **blkjmp**, the function that issued the block receives control with the new signal mask, and it is not required to know whether it received control as the result of a regular **longjmp** or a **siglongjmp**.

Note: The **sigjmp_buf** data type contains more information than a standard **jmp_buf**. For this reason, you cannot call **siglongjmp** with a buffer filled by the **setjmp** function.

RETURN VALUE

siglongjmp does not return a value.

CAUTION

The function in which the corresponding call to **sigsetjmp** is made must not have returned before you make the call to **siglongjmp**. If **siglongjmp** is passed a **value** of 0, a 1 is substituted.

EXAMPLE

See the example for **sigsetjmp**.

RELATED FUNCTIONS

longjmp, **sigsetjmp**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

signal Define Program Signal Handling

ISO ANSI UNIX

SYNOPSIS

```
#include <signal.h>

   /* This typedef is in <signal.h>. */
typedef void (*_HANDLER)(int);

_HANDLER signal(int signum, _HANDLER handler);
```

DESCRIPTION

signal defines the action taken when a signal is received by the program. The **signum** argument is the number of the signal, which should be specified as a symbolic signal name. Refer to the signal names listed in "Supported Signals" on page 5-3 .

The **handler** argument is the address of the function to be called when the signal occurs. The **handler** argument can point to a user function, or it can specify one of the two symbolic values SIG_DFL or SIG_IGN. If you specify SIG_IGN, the signal is ignored, if possible; if you specify SIG_DFL, the default action for the signal is taken. (Note that for most signals, the default action is program termination.) Details of what occurs when you specify SIG_DFL or SIG_IGN are provided in the descriptions of the signals. In addition, Tables 5.2 and 5.3 summarize default actions and the results of ignoring signals.

Refer to "Handling Signals" on page 5-5 for a detailed description of how to use **signal**.

RETURN VALUE

signal returns the address of the previous handler for the signal. If the signal was previously ignored, SIG_IGN is returned; if no action was defined for the signal, SIG_DFL is returned. If the call to **signal** cannot be honored (for example, if you specify SIG_IGN for a signal that cannot be ignored), the special value SIG_ERR is returned.

CAUTION

When the library discovers a signal with a handler defined by **signal**, it first restores default signal handling with the following call before it executes the handler you have defined:

```
signal(signum, SIG_DFL);
```

Therefore, it is necessary to reissue **signal** to handle a recurrence of the same signal.

PORTABILITY

The details of signal handling vary widely from system to system. See "Using Signals Portably" on page 5-18 for information on the portable use of signals.

signal Define Program Signal Handling
(continued)

EXAMPLE

```
#include <lcsignal.h>
#include <float.h>

signal(SIGFPOFL, &overflow_handler);

   /* Perform calculation. */
.
.
.
   /* This function handles a floating-point overflow */
   /* by replacing the result of the computation with */
   /* plus or minus DBL_MAX and allowing the          */
   /* computation to continue.                        */
   /* This example assumes that SIGFPE is not an      */
   /* OpenEdition-managed signal.                     */

void overflow_handler(int signum)
{
   FPE_t *info;
   info = siginfo(); /* Get information about signal. */
   if (!info)

         /* If no information is available, force     */
         /*  default handling.                        */
      raise(SIGFPOFL);

      /* Replace result by appropriate large number.  */
   if (*info->result.doublev < 0.0)
      *info->result.doublev = -DBL_MAX;

   else
      *info->result.doublev = DBL_MAX;
}
```

RELATED FUNCTIONS

sigaction, **cosignal**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

sigpause Suspend Execution and Control Blocking of Signals

SYNOPSIS

```
#include <lcsignal.h>

int sigpause(int mask);
```

DESCRIPTION

sigpause temporarily changes the signal mask for asynchronous signals and suspends program execution until an unblocked signal is discovered. When the program resumes, the signal mask is restored to its previous settings. The **mask** argument is an integer interpreted as a bit string. You use this bit string to set the mask of blocked signals. If program execution is suspended and a signal occurs that is blocked by the mask, the program does not resume.

The most common use of this function is to unblock all signals while program execution is suspended. For example, **sigpause(0)** suspends execution until any signal occurs and then restores the previous mask to block whatever signals were blocked before the pause began.

When a signal that is not blocked by the mask is discovered, execution of the program resumes, the signal mask in effect when **sigpause** was called is restored, and the handler for the signal is called. Because **sigpause** restores the previous mask when the pause ends, you can use **sigpause** to handle a single occurrence of a signal, even if more than one signal is pending.

Note: If the mask that **sigpause** restores unblocks any signals that were blocked by the call to **sigpause**, you may encounter problems when several signals of different types are pending. Refer to CAUTION.

RETURN VALUE

sigpause returns the **errno** value EINTR, which indicates that it was interrupted by a signal. Refer to "The errno Variable" on page 1-7 for more information on **errno**.

CAUTION

If your program uses OpenEdition signals, use **sigsuspend** rather than **sigpause**. The **sigpause** function does not allow you to control the blocking of OpenEdition signals.

The most common use of **sigpause** is to specify **sigpause(0)**, which allows any signal to be handled. If you specify an argument for **sigpause** other than 0, avoid blocking a signal in the mask that is not blocked when you call **sigpause**. The reason for this caution is that when an unblocked signal occurs, the old mask is restored, which unblocks pending signals. If there are pending signals, they may interrupt the handler for the original signal, causing considerable confusion.

For example, suppose that SIGINT signals are not blocked and you call **sigpause(1<<(SIGINT-1))**. This call causes SIGINT signals to be blocked before execution is suspended. Suppose that execution resumes because SIGALRM occurs and that a SIGINT signal also occurs. Before the handler for SIGALRM is called, **sigpause** restores the original mask. This means the pending SIGINT signal can now be discovered during the execution of the handler for SIGALRM; the program must be prepared for this possibility.

sigpause Suspend Execution and Control Blocking of Signals
(continued)

IMPLEMENTATION

sigpause is implemented using idle waiting; that is, no CPU time is consumed (other than set-up time) while **sigpause** is executing.

EXAMPLE

```
#include <lcsignal.h>

volatile int shutdown = 0;

   /* SIGINT signals are blocked while the SIGINT handler */
   /*  executes.                                          */
signal(SIGINT, &int_handler);
sigblock(1<<(SIGINT-1));

   /* Wait for and handle interrupts one at a time. It is */
   /*  assumed that the SIGINT handler sets shutdown to   */
   /*  nonzero to cause program termination.              */
while(!shutdown)
   sigpause(0);
```

RELATED FUNCTIONS

ecbpause, **pause**, **sigsuspend**, **sleep**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

sigpending Return Pending Signals

SYNOPSIS

```
#include <lcsignal.h>

int sigpending(sigset_t *set);
```

The synopsis for the POSIX implementation is

```
#include <signal.h>

int sigpending(sigset_t *set);
```

You should use **`<signal.h>`** only if an appropriate feature test macro has been defined.

DESCRIPTION

`sigpending` returns the signals that are currently pending. The signal numbers are stored in the signal set addressed by **`set`**.

The **`sigpending`** function tests for pending signals (signals that have been generated for the process, but not delivered). In a POSIX system without extensions, a signal can be pending only if the signal is blocked. Some SAS/C extensions can delay delivery of one or more signals, even though the signal is not blocked. Any such delayed signals are included in the set of pending signals stored by **`sigpending`**.

RETURN VALUE

`sigpending` returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

```
#include <signal.h>
#include <stdio.h>

   /* Define structure of POSIX signal names and numbers. */
const struct {
   int signum;
   char *signame;
} sigtable[] = {
   {SIGABRT, "ABRT"},
   {SIGALRM, "ALRM"},
   {SIGCHLD, "CHLD"},
   {SIGCONT, "CONT"},
   {SIGFPE,  "FPE"},
   {SIGHUP,  "HUP"},
   {SIGILL,  "ILL"},
   {SIGINT,  "INT"},
   {SIGKILL, "KILL"},
   {SIGPIPE, "PIPE"},
   {SIGQUIT, "QUIT"},
   {SIGSEGV, "SEGV"},
   {SIGSTOP, "STOP"},
```

sigpending Return Pending Signals

(continued)

```
      {SIGTERM, "TERM"},
      {SIGTSTP, "TSTP"},
      {SIGTTIN, "TTIN"},
      {SIGTTOU, "TTOU"},
      {SIGUSR1, "USR1"},
      {SIGUSR2, "USR2"}};

   void show_pending(void) {
      sigset_t sigset;
      int i;
      int count;

      if (sigpending(&sigset) != 0)
         perror("sigpending error");
      else {
         count = 0;            /* Initialize pending count.  */
         for(i = 0; i < sizeof(sigtable)/sizeof(sigtable[0]); ++i)
            if (sigismember(&sigset, sigtable[i].signum)) {
               printf("Signal SIG%s is pending.", sigtable[i].signame);
               ++count;
            }
         if (count == 0)       /* if no signals were pending */
            puts("No POSIX signals are pending.");
      }
   }
```

RELATED FUNCTIONS

sigaddset, **sigprocmask**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

sigprocmask Modify a Program's Signal Mask

SYNOPSIS

```
#include <lcsignal.h>

int sigprocmask(int block, const sigset_t *newset, sigset_t *oldset);
```

The synopsis for the POSIX implementation is

```
#include <signal.h>

int sigprocmask(int block, const sigset_t *newset, sigset_t *oldset);
```

You should use **<signal.h>** only if an appropriate feature test macro has been defined.

DESCRIPTION

sigprocmask modifies the signal mask of the calling program.

block specifies the type of modification. Values for **block** are

SIG_BLOCK	specifies that the signals in **newset** should be blocked (other signals are not changed).
SIG_UNBLOCK	specifies that the signals in **newset** should not be blocked (other signals are not changed).
SIG_SETMASK	specifies that exactly the set of signals specified by **newset** should be blocked.

The **newset** and **oldset** arguments are both pointers to structures of type **sigset_t**, which is declared in **<lcsignal.h>** and **<signal.h>**. The **newset** argument is the new set of signals that should be blocked or unblocked. If **newset** is NULL, then the mask is not changed. **oldset** is a pointer to a signal set where the previous set of blocked signals is to be stored. If **oldset** is NULL, the previous signal mask is not stored.

You can use the **sigprocmask** function to change the blocking of signals that are managed by either SAS/C or OpenEdition. If **oesigsetup** has not been called, any attempt to change the blocking status of signals managed by OpenEdition is ignored.

RETURN VALUE

sigprocmask returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

See the example for **sigsetjmp**.

RELATED FUNCTIONS

sigaddset, **sigsuspend**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1

"Signal-Handling Functions" on page 2-17

sigsetjmp Save Stack Environment and Signal Mask

SYNOPSIS

```
#include <lcjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

The synopsis for the POSIX implementation is

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
```

You should use **`<setjmp.h>`** only if an appropriate feature test macro has been defined.

DESCRIPTION

`sigsetjmp` saves the current stack environment and signal mask. **`env`** is a pointer to the **`sigjmp_buf`** structure that contains the current environment. **`savemask`** specifies whether or not the signal mask is saved; if **`savemask`** is 0, the current signal mask is not saved. A nonzero value specifies that the current signal mask is saved.

RETURN VALUE

`sigsetjmp` returns the value 0, or the value specified in the call to **`siglongjmp`**.

IMPLEMENTATION

`sigsetjmp` is defined as a macro and should not be used with **`#undef`**.

EXAMPLE

The following example is an outline of an application using **`sigsetjmp`** and **`siglongjmp`** for error recovery. The program is designed so that if a protection exception (**`SIGSEGV`**) occurs, control is returned using **`siglongjmp`** to any of several locations, depending on when the error occurred. The signal mask at the time of the exception may differ from the signal mask of the cleanup code, which is unknown.

```
   /* This example must be compiled with the posix          */
   /* compiler option.                                      */
#include <sys/types.h>
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>
#include <unistd.h>
#include <lclib.h>

   /* This is the address of a sigjmp buffer defining        */
   /* the SIGSEGV                                            */
   /*  recovery location.                                    */
```

sigsetjmp Save Stack Environment and Signal Mask
(continued)

```
sigjmp_buf *recover;
extern void reinit(void);
extern void fixfiles(void);
extern void int_handler(int);   /* SIGINT handler - not shown */
extern void alrm_handler(int);  /* SIGALRM handler - not shown*/
extern void term_handler(int);  /* SIGTERM handler - not shown*/
static void segv_handler(int);  /* SIGSEGV handler            */

   /* shutdown flag, set by SIGTERM handler */
int shutdown = 0;

main() {
   sigjmp_buf mainbuf;
   struct sigaction int_action, alrm_action, segv_action;
   sigset_t newmask, oldmask;

      /* This code defines the normal recovery action, which  */
      /* is to reinitialize the program.  No signals are      */
      /* masked at this  point.                               */
   if (sigsetjmp(mainbuf, 1)) {   /* if a SIGSEGV occurred    */
      fputs("SIGSEGV error, reinitializing...", stderr);
      fflush(stderr);
      reinit();
   }
   recover = &mainbuf;            /* Save recovery buffer.    */

      /* Define signal actions.                               */
   segv_action.sa_handler = &segv_handler;
   segv_action.sa_flags = 0;
      /* Block SIGINT, SIGALRM and SIGTERM while we're        */
      /* handling SIGSEGV.                                    */
   sigemptyset(&segv_action.sa_mask);
   sigaddset(&segv_action.sa_mask, SIGINT);
   sigaddset(&segv_action.sa_mask, SIGALRM);
   sigaddset(&segv_action.sa_mask, SIGTERM);
   sigaction(SIGSEGV, &segv_action, NULL);

   alrm_action.sa_handler = &alrm_handler;
   alrm_action.sa_flags = 0;
      /* Block SIGINT while we're handling SIGALRM.           */
   sigemptyset(&alrm_action.sa_mask);
   sigaddset(&alrm_action.sa_mask, SIGINT);
   sigaction(SIGALRM, &alrm_action, NULL);

   int_action.sa_handler = &int_handler;
   int_action.sa_flags = 0;
      /* Block SIGALRM while we're handling SIGINT.           */
   sigemptyset(&int_action.sa_mask);
   sigaddset(&int_action.sa_mask, SIGALRM);
   sigaction(SIGINT, &int_action, NULL);

   term_action.sa_handler = &term_handler;
```

sigsetjmp Save Stack Environment and Signal Mask
(continued)

```
   term_action.sa_flags = 0;
      /* Only SIGTERM is blocked in the SIGTERM handler.            */
   sigemptyset(&term_action.sa_mask);
   sigaction(SIGTERM, &term_action, NULL);

   sigemptyset(&newmask); /* Set up mask to block SIGINT + SIGALRM. */
   sigaddset(&newmask, SIGINT);
   sigaddset(&newmask, SIGALRM);

   alarm(5);              /* Perform checkpoint every 5 seconds.    */

   while(!shutdown) {
      sigjmp_buf updbuf;  /* recovery buffer for update code        */
      sigjmp_buf *old_recover;   /* previous recovery buffer        */
         /* This part of the application reads input from stdin ... */
            .
            .
            .
         /* This part of the program updates a database and is      */
         /*  protected from SIGINT and SIGALRM interrupts.          */
      sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* Block signals. */
      old_recover = recover; /* Save previous recovery information. */
      if (sigsetjmp(updbuf, 1)) {    /* error during update         */
         fputs("SIGSEGV during database update - fixing filesn",
               stderr);
         fixfiles();
         recover = old_recover;
         sigprocmask(SIG_SETMASK, &oldmask, 0);  /* Restore mask.   */
         continue;
      }
      recover = &updbuf;  /* Define new recovery point.             */
         .
         .                /* Update the database.                 */
         .
      recover = old_recover;   /* Restore old recovery point.       */
         /* Restore the mask.                                       */
      sigprocmask(SIG_SETMASK, &oldmask, NULL);
   }
}

static void segv_handler(int signum) {
      /* This routine handles SIGSEGV errors by escaping to the     */
      /*  cleanup routine identified by recover.  The handler has   */
      /*  SIGALRM, SIGINT and SIGTERM signals blocked.  The mask    */
      /*  to be used during recovery is unknown.                    */

   btrace(0);            /* Get traceback for diagnosis of problem. */
   siglongjmp(*recover, 1);   /* Escape to cleanup routine.         */
}
```

sigsetjmp Save Stack Environment and Signal Mask
(continued)

RELATED FUNCTIONS

`setjmp`, **`siglongjmp`**

SEE ALSO

Chapter 5, "Signal-Handling Functions" on page 5-1
"Signal-Handling Functions" on page 2-17

sigsetmask Inhibit or Permit Discovery of Signals

SYNOPSIS

```
#include <lcsignal.h>

int sigsetmask(int mask);
```

DESCRIPTION

sigsetmask blocks or unblocks one or more asynchronous signals. The **mask** argument is an integer interpreted as a bit string, with each bit corresponding to a particular signal. You use this bit string to specify a mask of blocked signals.

For example, the following call blocks the SIGALRM signal and unblocks all other signals:

```
sigsetmask(1<<(SIGALRM - 1));
```

You can use the same format to block any single asynchronous signal; simply change the name of the signal to be blocked. To unblock all signals, use **sigsetmask(0)**.

The SAS/C library honors only bits corresponding to the asynchronous signals (SIGINT, SIGALRM, SIGIUCV, and SIGASY1 through SIGASY8); any other bits set in the mask are ignored. Also, **sigsetmask** does not affect any signals managed by OpenEdition. For this reason, **sigprocmask**, which can be used for all signals, is preferable to **sigsetmask**.

If a signal occurs while it is blocked, the signal is kept pending until it is unblocked by a call to **sigsetmask** or **sigpause**. When the signal is unblocked, it is discovered, and the appropriate handler is called. When a program begins executing, no signals are blocked. Refer to "Blocking Signals" on page 5-14 for more information.

For compatibility with previous releases, a call to **sigsetmask** requesting that all signals be blocked (a signal mask of all ones) causes all blockable OpenEdition signals to be blocked as well. This blocking occurs within the library, so if you call **sigbsetmask(0xffffffff)** and then use an **exec** function to transfer control to another program, that program receives control with no signals blocked.

RETURN VALUE

sigsetmask returns the previous mask of blocked signals. You can pass this value to **sigsetmask** later to restore the previous mask. Bits of the mask corresponding to synchronous signals are always 0.

CAUTION

You should not keep signals blocked for long periods of time because this may use large amounts of memory to queue pending signals. For lengthy programs, you should use **sigblock** to protect critical sections of the program and then reset the mask with **sigsetmask** to allow signals to occur freely in less critical areas.

The library sometimes blocks signals to delay asynchronous signals during its own processing. If the library is in the middle of processing and something occurs that causes it to call **longjmp** to return to your program, the mask set by the library may still be in effect; that is, the mask may not be what you specified in your program. For example, suppose a library function runs out of

sigsetmask Inhibit or Permit Discovery of Signals
(continued)

stack space and raises SIGMEM, and the handler for SIGMEM returns to your program with a **`longjmp`**. You may need to issue **`sigsetmask`** at the completion of the jump to restore the signal mask needed by the program. The functions **`sigsetjmp`** and **`siglongjmp`** may be useful in these situations.

A signal generated by the program calling **`raise`** or **`siggen`** always occurs immediately, even if the signal is blocked.

PORTABILITY

`sigsetmask` is only portable to BSD-compatible UNIX operating systems.

EXAMPLE

Refer to the example for **`sigblock`**.

RELATED FUNCTIONS

`sigblock`, **`sigpause`**, **`sigprocmask`**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

sigsuspend Replace Signal Mask and Suspend Execution

SYNOPSIS

```
#include <signal.h>

int sigsuspend(const sigset_t *set);
```

DESCRIPTION

sigsuspend replaces the signal mask for a program with the set of signals addressed by **set** and suspends program execution until an unblocked signal is received. If there is no handler defined for that signal, the default signal action (usually abnormal termination) is performed. Otherwise, the signal mask in effect when **sigsuspend** was called is restored before the handler is called.

The most common use of this function is to unblock all signals while program execution is suspended. For example, the following code suspends execution until any signal occurs and then restores the previous mask to block whatever signals were blocked before **sigsuspend** was called

```
sigemptyset(&mask);
sigsuspend(&mask);
```

Because **sigsuspend** restores the previous mask when a signal is discovered, you can use this function to handle a single occurrence of a signal, even if several signals are pending.

RETURN VALUE

sigsuspend never returns unless interrupted by a signal. If **sigsuspend** returns, it returns −1.

EXAMPLE

```
#define _SASC_POSIX_SOURCE 1

#include <signal.h>
#include <stdio.h>

volatile int shutdown = 0;
struct sigaction int_action;
sigset_t blocked_set, empty_set;

int_action.sa_handler = &int_handler;
sigemptyset(&int_action.sa_mask);
int_action.flags = 0;
sigaction(SIGINT, &int_action, NULL);
sigemptyset(&blocked_set);
sigaddset(&blocked_set, SIGINT);
sigprocmask(&blocked_set);
sigemptyset(&empty_set);
```

sigsuspend Replace Signal Mask and Suspend Execution

(continued)

```
      /* Wait for and handle interrupts one at a time. It  */
      /*  is assumed that the SIGINT handler sets shutdown */
      /*  to nonzero to cause program termination.        */
   while(!shutdown)
      sigsuspend(&empty_set);
```

RELATED FUNCTIONS

`ecbsuspend`, **`sleep`**

SEE ALSO

Chapter 5, “Signal-Handling Functions” on page 5-1

“Signal-Handling Functions” on page 2-17

sin Compute the Trigonometric Sine

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double sin(double x);
```

DESCRIPTION

sin computes the trigonometric sine of its argument **x** expressed in radians. Because the sine function is periodic, only the value of x mod 2π is used to compute the sine. If **x** is very large, only a limited precision is left to represent x mod 2π. Thus, an error message is written for very large negative or positive arguments (see DIAGNOSTICS).

RETURN VALUE

sin returns the principal value of the sine of the argument **x**, if this value is defined and computable. The return value is of type **double**.

DIAGNOSTICS

For a very large argument (**x** > 6.7465e9), the function returns 0.0. In this case, the message “total loss of significance” is also written to **stderr** (the standard error file).

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

EXAMPLE

This example computes the trigonometric cosecant of a value:

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

#define SVECTOR .7854

main()
{
   double cosec;

      /* The cosecant of a value is 1 divided */
      /*  by the sine.                        */
   cosec = 1 / sin(SVECTOR);
   printf("1 / sin(%f) = %f\n", SVECTOR, cosec);
}
```

RELATED FUNCTIONS

cos, **_matherr**, **tan**

SEE ALSO

“Mathematical Functions” on page 2-6

sinh Compute the Hyperbolic Sine

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double sinh(double x);
```

DESCRIPTION

sinh computes the hyperbolic sine of its argument **x**, expressed by this relation:

$$r = (e^{x} - e^{-x})/2$$

RETURN VALUE

sinh returns the principal value of the hyperbolic sine of the argument **x**, if this value is defined and computable. The return value is a double-precision, floating-point number.

DIAGNOSTICS

For a positive value of **x** that is too large, the **sinh** function returns **HUGE_VAL**. For a negative **x** value that is too large, **sinh** returns **-HUGE_VAL**. In both cases, the run-time library writes an error message to **stderr** (the standard error file).

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

EXAMPLE

This example computes the hyperbolic cosecant of a value:

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

#define YRANG 1.30

main()
{
   double cosec_h;

      /* The hyperbolic cosecant of a value is 1      */
      /* divided by the hyperbolic sine of the value. */
   cosec_h = 1 / sinh(YRANG);
   printf("1 / sinh(%f) = %f\n", YRANG, cosec_h);
}
```

RELATED FUNCTIONS

cosh, **_matherr**, **tanh**

SEE ALSO

"Mathematical Functions" on page 2-6

sleep, sleepd Suspend Execution for a Period of Time

POSIX UNIX

SYNOPSIS

```
#include <lclib.h>

unsigned sleep(unsigned sec);
double sleepd(double sec);
```

The synopsis for the POSIX implementation is

```
#include <sys/types.h>
#include <unistd.h>

unsigned sleep(unsigned sec);
double sleepd(double sec);
```

DESCRIPTION

sleep and **sleepd** are called to suspend execution of the program for a specified number of seconds or until a signal occurs. If the value of the **sec** argument is 0, the **sleep** or **sleepd** function immediately returns to its caller. The behavior of the **sleep** and **sleepd** functions depends on whether **SIGALRM** is defined as a signal managed by SAS/C or an OpenEdition signal. If **SIGALRM** is managed by OpenEdition, the **sleep** function is implemented by OpenEdition, and the **sleepd** function is not implemented. In this case, note that the occurrence of a signal managed by SAS/C does not cause **sleep** to terminate. If **SIGALRM** is managed by SAS/C, a value of **sec** that is more than a day (86,400 seconds) is treated as a day.

You may use **sleep** or **sleepd** in association with the **alarm** or **alarmd** function. If either of these functions is called before completion of a time interval set with an alarm function, execution resumes when the alarm interval ends or the sleep interval ends, whichever occurs first.

The SAS/C implementation of **sleep** and **sleepd** always unblocks the **SIGALRM** signal to enable them to detect the completion of the time interval. However, no other signals are unblocked, and the signal mask is completely restored before these functions return. If a signal is raised and then blocked, program execution does not resume. If an unblocked signal occurs, the handler for the signal is executed before these functions return to the program that called it.

sleepd performs the same actions as **sleep** but permits the amount of time to be specified with greater accuracy. The accuracy of timing depends on the operating system and CPU model.

RETURN VALUE

If the sleep period ends because the specified time has elapsed, the **sleep** and **sleepd** functions return 0. If a signal occurs that ends the sleep period, the functions return the amount of time left in the sleep interval; **sleep** rounds up to an integer of seconds.

sleep, sleepd

(continued)

Suspend Execution for a Period of Time

CAUTION

Under a non-XA, non-ESA version of CMS, you must use the CP command SET TIMER REAL for proper functioning of **sleep** and **sleepd**. If SET TIMER REAL is not in effect, a diagnostic message is produced and these functions return immediately.

PORTABILITY

sleepd is not portable.

IMPLEMENTATION

sleep and **sleepd** are implemented using idle waiting; that is, no CPU time is consumed (other than set-up time) during the sleep interval.

EXAMPLE

This example calls the routine **acquire** to get exclusive control of a file. It tries to obtain control four times a second until it is successful.

```
#include <lclib.h>

   /* Return 1 if successful,and 0 if unsuccessful. */
int acquire();

while (!acquire())
   sleepd(0.25);
.
.
.
```

RELATED FUNCTIONS

alarm, **alarmd**, **select**, **sigsuspend**

SEE ALSO

"SIGALRM" on page 5-23

"Signal-Handling Functions" on page 2-17

snprintf Write a Limited Portion of Formatted Output to a String

SYNOPSIS

```
#include <lcio.h>

int snprintf(char *dest, size_t maxlen, const char *format,
             var1, var2, ...);
```

DESCRIPTION

The **snprintf** function writes formatted output to the area addressed by **dest** under control of the string addressed by **format** until either all format conversion specifications have been satisfied, or **maxlen** characters have been written. The **snprintf** function is equivalent to the **sprintf** function, except that no more than **maxlen** characters are written to the **dest** string.

If the **maxlen** limit is reached

- a terminating-null character is not added
- the number of characters placed in the output area are the value of **maxlen**
- the remainder of the format string is ignored
- the **snprintf** function returns a negative value whose magnitude is equal to the value of **maxlen**.

In all other respects, **snprintf** behaves identically to **sprintf**. The string pointed to by **format** is in the same form as that used by **fprintf**. Refer to the **fprintf** function description for detailed information concerning format conversions.

RETURN VALUE

The **snprintf** function returns an integer value that equals, in magnitude, the number of characters written to the area addressed by **dest**. If the value returned is negative, then either the **maxlen** character limit was reached or some other error, such as an invalid format specification, has occurred. The one exception to this is if an error occurs before any characters are stored, **snprintf** returns **INT_MIN (-2**31)**.

CAUTION

If the **maxlen** value is 0, no characters are written, and **snprintf** returns 0. If the value is greater than **INT_MAX**, then **snprintf** behaves identically to **sprintf**, in that no limit checking is done on the number of characters written to the output area.

No warnings concerning length errors are produced by **snprintf**, and the only indication that the output may have been truncated or is incomplete is a negative return value.

IMPLEMENTATION

The **snprintf** function, when invoked with a limit greater than 512 characters, calls the **malloc** function to obtain a temporary spill buffer equal in size to the limit specified. If insufficient storage is available, **snprintf** attempts to process the format specifications with an internal 512-byte spill buffer. In this case, individual conversion specifiers that produce more than 512 characters may fail, and **snprintf** processing can terminate prematurely.

snprintf Write a Limited Portion of Formatted Output to a String
(continued)

EXAMPLE

This example writes out the first 5 lines of a file. If the lines are longer than the program's output buffer, they are truncated:

```
#include <stdlib.h>
#include <lcio.h>

#define BUFFER_SIZE 40
#define LIMIT (BUFFER_SIZE - sizeof("..."))

char *_style = "tso";

main()
{
   char fname[80];
   char inbuf[300];
   char buffer[BUFFER_SIZE];
   FILE *input;
   int i;
   int count;

   puts("Enter the name of an input file.");
   gets(fname);
   input = fopen(fname, "r");
   if (!input) {
      puts("File could not be opened.");
      exit(EXIT_FAILURE);
   }
   for (i = 0; i < 5; ++i) {
      if (!fgets(inbuf, 300, input))  /* end of file         */
         break;
      count = snprintf(buffer, LIMIT,
                       "Line %d of file is : %s", i, inbuf);
      if (count == LIMIT) buffer [LIMIT] = '\0';
                                      /* Output fit exactly. */

      else if (count == -LIMIT)       /* output truncated    */
         strcpy(buffer+LIMIT, "...");
      puts(buffer);
   }
   fclose(input);
   exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

sprintf, **vsnprintf**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

sprintf Write Formatted Output to a String

ISO ANSI UNIX

SYNOPSIS

```
#include <stdio.h>

int sprintf(char *dest, const char *format,
            var1, var2, ...);
```

DESCRIPTION

`sprintf` writes formatted output to the area addressed by **`dest`** under the control of the string addressed by **`format`**. In the argument list following **`format`**, there may be one or more additional arguments whose values are to be formatted and transmitted.

The string pointed to by **`format`** is in the same form as that used by **`fprintf`**. Refer to the **`fprintf`** description for detailed information concerning the formatting conversions.

RETURN VALUE

`sprintf` returns the number of characters written to the area addressed by **`dest`**.

CAUTION

Overruns of the destination area cannot be detected or avoided by **`sprintf`**. Thus, you must ensure that the destination area is large enough.

IMPLEMENTATION

`sprintf` is just like **`fprintf`**, with two exceptions:

- No file output is performed. Instead, the formatted text is stored in the area addressed by **`dest`**.
- There is a maximum of $2^{24}-1$ characters (16M−1) produced per conversion.

sprintf Write Formatted Output to a String
(continued)

EXAMPLE

This example transforms a list of names in the form “first, middle, last” into the form “last, first, middle”:

```
#include <stdio.h>
#include <string.h>

char *names[] = {
   "John M. Brown",
   "Daniel Lopez",
   "H. Margaret Simmons",
   "Ralph Jones",
   "Harry L. Michaels"
};

main()
{
   char lfm[94];
   char first[31], last[31], middle[31];
   int i, n;

   puts("The names in f-m-l format are:");
   for (i = 0; i < sizeof(names)/sizeof(names[0]); ++i)
      puts(names[i]);
   puts("\nThe names in l-f-m format are:");
   for (i = 0; i < sizeof(names)/sizeof(names[0]); ++i) {
      n = sscanf(names[i], "%s %s %s", first, middle, last);
      if (n != 3){          /* There was no middle name. */
         strcpy(last, middle);
         middle[0] = '\0';
      }
      sprintf(lfm, "%s, %s %s", last, first, middle);
      puts(lfm);
   }
}
```

RELATED FUNCTIONS

format, **fprintf**, **sprintf**, **vsprintf**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

sqrt Compute the Square Root

ISO ANSI UNIX

SYNOPSIS

```
#include <math.h>

double sqrt(double x);
```

DESCRIPTION

sqrt computes the square root of its argument **x**.

RETURN VALUE

sqrt returns the positive square root of **x**, expressed as a double-precision, floating-point number.

DIAGNOSTICS

For a negative value of **x**, the function returns 0.0, and the run-time library writes an error message to **stderr** (the standard error file).

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

EXAMPLE

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

main()
{
   double x;
   puts("Enter the number you want the square root of: ");
   scanf("%lf", &x);
   printf("The square root of %f is %f\n", x, sqrt(x));
}
```

RELATED FUNCTIONS

hypot, **_matherr**

SEE ALSO

"Mathematical Functions" on page 2-6

srand Initialize Random Number Generator

ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

void srand(unsigned int seed);
```

DESCRIPTION

srand resets the number generator to a new starting point. The **rand** function then uses this **seed** to generate a sequence of pseudorandom numbers. The initial default **seed** is 1.

RETURN VALUE

srand has no return value.

PORTABILITY

See the portability details for **rand** for more information.

EXAMPLE

This example uses **srand** to print 1,000 random numbers:

```
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
   int i, x;

   if (argc > 1){
      x = atoi(argv[i]);
      if (x == 0){
         x = 1;
      }
      printf("Seed value is %d\n",x);
      srand(x);
   }
   puts("Here are 1000 random numbers:");
   for (i = 0; i < 200; i++){
      printf("%5d %5d %5d %5d %5d\n",
             rand(),rand(),rand(),rand(),rand());
   }
   puts("\n");
}
```

RELATED FUNCTIONS

rand

SEE ALSO

"Mathematical Functions" on page 2-6

sscanf Read Formatted Data from a String

SYNOPSIS

```
#include <stdio.h>

int sscanf(const char *source, const char *format,
           loc1, loc2, ... );
```

DESCRIPTION

`sscanf` reads formatted input text from the string addressed by **`source`**. No file input is performed. Following the format in the argument list may be one or more additional pointers addressing storage where the input values are stored.

The string pointed to by **`format`** is in the same form as that used by **`fscanf`**. Refer to the **`fscanf`** description for detailed information concerning the formatting conventions.

RETURN VALUE

`sscanf` returns **`EOF`** if end of string (or failure to match) 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 **`sscanf`** argument pointers. Attempting to read past the null terminator of the **`source`** string is treated like an end of file on the **`fscanf`** input file.

IMPLEMENTATION

`sscanf` is just like **`fscanf`**, except that input data are taken from a string rather than a file.

If **`sscanf`** encounters an error in storing input values, it stores the values up to the error and then stops.

EXAMPLE

`sscanf` is illustrated in the example for **`sprintf`**.

RELATED FUNCTIONS

`fscanf`, **`scanf`**, **`strtod`**, **`strtol`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

stat Determine File Status by Pathname

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(char pathname, struct stat *info);
```

DESCRIPTION

stat gets status information for an OpenEdition HFS file and returns it in a **stat** structure, defined in **<stat.h>**. The file pathname is **pathname**. You must specify the pathname as an OpenEdition HFS file. For programs not compiled with the **posix** option, a style prefix may be required. See "File Naming Conventions" on page 3-56 for information on specifying OpenEdition filenames. **info** is the area of memory in which the status information is stored. The **<sys/stat.h>** header file contains a collection of macros that you can use to examine properties of a **mode_t** value from the **st_mode** field. See the **fstat** function for information about these macros.

RETURN VALUE

stat returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

The following example is a function that you can call to determine whether two pathnames represent the same file. (Two different pathnames might represent the same file due to the use of links, or the use of **``.''** or **``..''** in the paths.) In this example, two different HFS files must have either different device numbers or different inode numbers:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int samefile(const char *path1, const char *path2) {
   struct stat stat1, stat2;
   int rc;
   rc = stat(path1, &stat1);
   if (rc == -1) {
      perror("stat error");
      return -1;
   }
   rc = stat(path2, &stat2);
   if (rc == -1) {
      perror("stat error");
      return -1;
   }
   if (stat1.st_dev == stat2.st_dev && stat1.st_ino == stat2.st_ino)
      return 1;
   else return 0;
}
```

stat Determine File Status by Pathname
(continued)

RELATED FUNCTIONS

`cmsstat`, `fattr`, `fstat`, `lstat`, `osddinfo`, `osdsinfo`

SEE ALSO

"File Management Functions" on page 2-16

stcpm Unanchored Pattern Match

SYNOPSIS

```
#include <lcstring.h>

int stcpm(char *str, const char *pat, char **substr);
```

DESCRIPTION

stcpm scans the input string addressed by **str** for the first occurrence of a substring matching the pattern addressed by **pat**. The **substr** function points to a word in which the first match to the search pattern is stored (if a match is found). This pointer is not used if no match is found. (See RETURN VALUE.)

You can specify the search pattern in several ways, possibly including the special characters *****, **?**, and **+**, as follows:

*	matches zero or more occurrences of the preceding character.
+	matches one or more occurrences of the preceding character.
?	matches any character.

- To scan for an exact match to a pattern, specify the pattern character for character. For example, to locate William A. Tell, specify **William A. Tell** as follows:

```
stcpm(str, "William A. Tell", substr);
```

- To match some elements of a pattern exactly, with other elements free to vary, use the question mark (**?**) for the elements that can vary:

```
stcpm(str, "William ?. Tell", substr);
```

This statement matches any pattern with William as the first item, any character as the middle initial, a period, and Tell as the last item. (If the middle initial is a question mark, it matches as well. More on this situation is discussed in the next item.)

- To match 0 or more occurrences of a character, use an asterisk (*****) in the search pattern. The asterisk should follow the character that may occur several times or not occur at all:

```
stcpm(str, "William A*. Tell", substr);
```

This statement matches the following list with any number of occurrences of A:

```
William . Tell
William A. Tell
William AAAAA. Tell
```

Similarly, this statement matches the list items that follow it:

```
stcpm(str, "William A*.* *Tell", substr);
```

stcpm Unanchored Pattern Match
(continued)

- a character sequence that contains a blank and no A **(`` '')**; a blank and one A **(`` A'')**; or a blank and several A's **(`` AAA'')**:

```
William  . Tell
William A. Tell
William AAAAA. Tell
```

- no period, one period, or several periods:

```
William A Tell
William A. Tell
William ..... Tell
```

- no blank, one blank, or several blanks before Tell:

```
William A.Tell
William A. Tell
William A.     Tell
```

- a combination of the possible variations:

```
William AAATell
William ...   Tell
William    Tell
```

As another example, the following statement matches a character sequence beginning with William and ending with Tell, with no middle initial character, or with any middle-initial character, including a question mark:

```
stcpm(str, "William ?*.* *Tell", substr);
```

To match *only* on a question mark, use a question mark preceded by a backslash. (Recall that the backslash is used as an escape character in string literals as well, so you must use two backslashes in a string literal to get one backslash in the pattern.)
For an exact match with William ?. Tell, use this statement:

```
stcpm(str, "William \\?. Tell", substr);
```

By comparison, no match is found for William A. Tell or for William . Tell when you use the search pattern in this statement.

- To match one or more occurrences of a character, use a plus sign (**+**) following the character in the search pattern:

```
stcpm(str, "William A. T+e+l+l+", substr);
```

stcpm Unanchored Pattern Match
(continued)

The search pattern in the above statement matches the following:

```
William A. Tell
William A. TTTeeelll
```

The matching continues with multiple occurrences of T, e, and l. Note that the plus sign matches one or more occurrences, but not zero occurrences of the character. (To find a match where there may be no occurrence of a character, use an asterisk (`*`) following the character.)

- To exactly match a plus sign (`+`) or asterisk (`*`), precede the `+` or `*` symbol in the search pattern with a backslash.

  You can combine the `+`, `*`, and `?` symbols in the search pattern when various combinations of characters are to be matched. Consider these examples:

```
   /* ?+ matches one or more question marks  */
char *pat1 = "William \\?+.* *Tell";

   /* ?* matches zero or more question marks */
char *pat2 = "William \\?*.* *Tell";
```

RETURN VALUE

`stcpm` returns the length of the character sequence that matches the search pattern, if successful, or 0 if no match is found. The character pointer that **`substr`** addresses points to the first match to the search pattern if a match is found.

CAUTION

The value returned by **`stcpm`** (**`str`**, **`pat`**, **`substr`**), the length of the character sequence that matches the search pattern, is not necessarily the same as the value returned by **`strlen(*substr)`** because **`strlen(*substr)`** returns the length of the input string from the beginning of the match to the null character that terminates the string. The length, in this case, may include characters that were not matched.

IMPLEMENTATION

For **`stcpm`**, the scan is not anchored. If no match occurs at the first position in the input string **`str`**, the next position is checked until a match is found, and so on until the input string is exhausted.

stcpm Unanchored Pattern Match
(continued)

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

main()
{
   static char *sps[ ]  = {
      "----William Tell----",    /* no middle initial         */
      "(((William A Tell)))",    /* middle initial            */
      "...William AAA. Teller",  /* middle initial and period */
      "As William S. Tell Jr"    /* wrong initial             */
   };

   char *reslt;
   char **q;
   int length, i;

   q = &reslt;

      /* Find William Tell, whether or not he used his        */
      /*  middle initial.                                      */
   for (i = 0; i < 4; i++) {
      if (length = stcpm(sps[i] , "William A*.* *Tell", q)){
         printf("\n%d. Match result =  ",i);
         fwrite(reslt, 1, length, stdout);
      }
      else
         printf("\n%d. No match for string = %s\n", i, sps[i] );
   }
}
```

RELATED FUNCTIONS

stcpma, **strchr**, **strstr**

SEE ALSO

"String Utility Functions" on page 2-3

stcpma Anchored Pattern Match

SYNOPSIS

```
#include <lcstring.h>

int stcpma(char *str, const char *pat);
```

DESCRIPTION

stcpma tests the input string addressed by **str** to determine whether it starts with a substring matching the pattern that **pat** points to. The search terminates if a match is not found at the beginning of the input string.

The pattern format can be specified using the symbols **+**, *****, and **?**, as described in the previous discussion of the **stcpm** function.

Although the **stcpm** and **stcpma** functions use the same pattern-matching notation, **stcpma** is different from **stcpm** in two ways:

- **stcpma** looks for a match to the search pattern (**pat**) only at the beginning of the input string (**str**). The **stcpm** function scans the entire string for the first match, which may or may not occur at the beginning.
- **stcpma** does not take a third argument.

RETURN VALUE

If it is successful, **stcpma** returns the length of the character sequence that matches the search pattern, or it returns 0 if no match is found.

In comparison to **stcpm**, both **stcpm** and **stcpma** can find the search pattern **William A*.* *Tell** in the following input string:

```
"William A. Tell, Margaret Fairfax-Tell"
```

However, **stcpma** cannot find the search pattern in this string:

```
"Margaret Fairfax-Tell, William A. Tell"
```

And **stcpma** cannot find it in this string, which contains an initial blank:

```
" William A. Tell"
```

stcpma Anchored Pattern Match
(continued)

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

main()
{
   static char *sps[ ]  = {
      "William Tellas  Sr.",           /* no middle initial  */
      "William A Tella      ",         /* middle initial     */
      "William AAA. Tell!! ---",       /* initial and period */
      "William S. Teller; Then..",     /* wrong initial      */
   };
   int length, i;

      /* Find William Tell, whether or not he used his       */
      /*  middle initial.                                    */
   for (i = 0; i < 4; i++) {
      if (length = stcpma(sps[i], "William A*.* *Tell"))
         printf("%d. Match result = %.*s\n", i,length, sps[i]);
      else
         printf("%d. No match for string = %s\n",i,sps[i]);
   }
}
```

RELATED FUNCTIONS

stcpm

SEE ALSO

"String Utility Functions" on page 2-3

strcat Concatenate Two Null-Terminated Strings

SYNOPSIS

```
#include <string.h>

char *strcat(char *to, const char *from);
```

DESCRIPTION

strcat adds characters from the second argument string **from** to the end of the first argument string **to** until a terminating-null character is found. The null also is copied.

RETURN VALUE

The return value is a pointer to the **to** string.

CAUTION

No check is made (or can be made) to see if there is room in the **to** string for all the characters of the combined string. Characters are copied until a null character is found in the source string, or until a protection or addressing exception occurs. A program check also can occur if the **to** string is not properly terminated. The effect of **strcat** is not defined if the **to** and **from** fields overlap.

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

#define MAXLINE 100

main()
{
   char line[MAXLINE+1];
   char message[MAXLINE+11];

   puts("Enter a line of text to be translated to lowercase letters:");
   gets(line);
   strcat(strcpy(message, "INPUT: "), line);
   puts(message); /* Label and echo the input.             */
   strlwr(line);  /* Turn the input into lowercase letters. */
   strcat(strcpy(message, "OUTPUT: "), line);
   puts(message); /* Label and print the results.          */
}
```

RELATED FUNCTIONS

strcpy, **strncat**

SEE ALSO

"String Utility Functions" on page 2-3

strchr Locate First Occurrence of a Character in a String

ISO ANSI UNIX

SYNOPSIS

```
#include <string.h>

char *strchr(const char *str, int ch);
```

DESCRIPTION

strchr searches the null-terminated string **str** for the first occurrence of a specific character **ch**, returning a pointer to the first occurrence, or **NULL** if the character does not occur in the string.

RETURN VALUE

The return value is a pointer to the first occurrence of the character in the argument string, or **NULL** if the character is not found. If the search character is the null character ('\0'), the return value addresses the null character at the end of the argument string.

CAUTION

A protection or addressing exception may occur if the argument string is not properly terminated.

See the **memscntb** function description for information on possible interactions between the **strchr**, **memscntb**, or **strscntb** functions.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

main()
{
   char *input;
   char *nl;

   input = "abcdefghijk\nlmnopqrstuvwxyz";
   if (nl = strchr(input, '\n')) *nl = '\0';
   printf("The new line character occurs after %c\n", *(nl-1));
}
```

RELATED FUNCTIONS

memchr, **stcpm**, **strpbrk**, **strrchr**, **strscan**, **strstr**

SEE ALSO

"String Utility Functions" on page 2-3

strcmp Compare Two Null-Terminated Strings

SYNOPSIS

```
#include <string.h>

int strcmp(const char *str1, const char *str2);
```

DESCRIPTION

strcmp compares two character strings (**str1** and **str2**) using the standard EBCDIC collating sequence. The return value has the same relationship to 0 as **str1** has to **str2**. If two strings are equal up to the point at which one terminates (that is, contains a null character), the longer string is considered greater.

RETURN VALUE

The return value from **strcmp** is 0 if the two strings are equal, less than 0 if **str1** compares less than **str2**, and greater than 0 if **str1** compares greater than **str2**. No other assumptions should be made about the value returned by **strcmp**.

CAUTION

If one of the arguments of **strcmp** is not properly terminated, a protection or addressing exception may occur. If one of the arguments to the built-in version of **strcmp** is a constant, the compiler generates a CLC instruction to perform the entire comparison. If the variable argument is not null terminated, the character-by-character comparison may perform as expected, but a comparison by the CLC instruction may cause an addressing exception in rare cases.

IMPLEMENTATION

The compiler generates inline code for **strcmp** unless **strcmp** is undefined (by an **#undef** statement) to prevent this. The inline code may still call a library routine in special cases.

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   char command[20];
   int n = 0;

   for(;;) {
      ++n;
      printf("Enter command # %d\n", n);
      puts("Enter quit to terminate/any other command to continue.");
      gets(command);
      if (strcmp(command, "quit") == 0) break;
```

strcmp Compare Two Null-Terminated Strings
(continued)

```
                /* Determine whether command is equal to quit. */
             strlwr(command);
             if (strcmp(command, "quit") == 0)
                exit(0);
             puts("Did you mean to say quit? (Case is significant.)");
          }
       }
```

RELATED FUNCTIONS

memcmp, **strcoll**, **strncmp**, **strxfrm**

SEE ALSO

"String Utility Functions" on page 2-3

strcpy Copy a Null-Terminated String

ISO ANSI UNIX

SYNOPSIS

```
#include <string.h>

char *strcpy(char *to, const char *from);
```

DESCRIPTION

strcpy copies characters from the second argument string, **from**, to the first argument string, **to**, until a terminating-null character is found. The null also is copied.

RETURN VALUE

The return value is a pointer to the **to** string.

CAUTION

No check is made (or can be made) to see if there is room in the **to** string for all the characters of the **from** string. Characters are copied until a null character is found, or until a protection or addressing exception occurs.

The effect of **strcpy** is not defined if the **to** and **from** fields overlap.

IMPLEMENTATION

Provided that **<string.h>** is included (by an **#include** statement) and **strcpy** is not undefined (by an **#undef** statement), **strcpy** is implemented by inline code.

EXAMPLE

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define WORDSIZE 20

main()
{
   char *text = "Some of this \"line\" is in \"quotes\".";
   char *quotepos1, *quotepos2;
   char word[WORDSIZE];

   printf("The input text is:\n%s\n", text);
   quotepos1 = strchr(text, ’"’);
   if (quotepos1 == NULL) {
      puts("There are no quotes in this line.");
      abort();
```

strcpy Copy a Null-Terminated String
(continued)

```
      }
      quotepos2 = strchr(quotepos1+1, '"');
      if (quotepos2 == NULL) {
         puts("There is only one quotation mark in this line.");
         abort();
      }
      if (quotepos2 - quotepos1 > WORDSIZE)
         puts("The first word in quotes is too large to handle.");
      else {
         strcpy(word, quotepos1+1);          /* Copy the word.           */
         word[quotepos2-quotepos1-1] = '\0'; /* Null-terminate the word. */
         printf("The first word in quotation marks in the text is \"%s\".",
                word);
      }
   }
```

RELATED FUNCTIONS

`memcpy`, **`strcat`**, **`strncpy`**, **`strsave`**

SEE ALSO

"String Utility Functions" on page 2-3

strcspn Locate the First Occurrence of the First Character in a Set

ISO ANSI UNIX

SYNOPSIS

```
#include <string.h>

size_t strcspn(const char *str, const char *cset);
```

DESCRIPTION

strcspn locates the first character in the first argument string (**str**) contained in the second argument string (**cset**), returning its position in the first argument.

RETURN VALUE

strcspn returns the number of consecutive characters not in the given character set found in the argument string, starting at the first character. If all characters of the string are not in the set (so that no character in the set can be found), the value returned is the length of the string. Similarly, if the character set is null (that is, if it contains no characters), the return value from **strcspn** is the length of the first argument.

CAUTION

A protection or addressing exception may occur if either argument is not properly terminated.

See the **memscntb** function description for information on possible interactions between the **strcspn**, **memscntb**, or **strscntb** functions.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

#define MAXLINE 100

main()
{
   char text[MAXLINE+1];
   size_t pos, len;
   int words;
   for (;;) {
      puts("Enter a line of text.");
      puts("Just press Enter to quit.");
      gets(text);
      if (text[0] == '\0') exit(0); /* Quit if null input.     */
      pos = 0;
      words = 0;
      for (;;) {
            /* Skip to next punctuation mark.                  */
         len = strcspn(text+pos, " .,?!");
```

strcspn Locate the First Occurrence of the First Character in a Set
(continued)

```
            /* if next character is punctuation                */
         if (len == 0) {
            ++pos;
            continue;              /* Skip to next character. */
         }
            /* encountered the end of the string */
         if (text[pos+len] == '\0') break;
         if (words == 0)
            puts("The words in the input line are:");
         ++words;
         do{
            putchar(text[pos]);
            ++pos;
         } while(--len);
         putchar('\n');
         ++pos;                    /* Skip the punctuation.   */
      }
   if (words == 0)
   puts("There were no words in that line.");
   }
}
```

RELATED FUNCTIONS

strrcspn, **strscan**, **strspn**

SEE ALSO

"String Utility Functions" on page 2-3

strerror Map Error Number to a Message String

SYNOPSIS

```
#include <string.h>

char *strerror(int errnum);
```

DESCRIPTION

strerror maps the error number in **errnum** to an error message string.

The message returned by **strerror** provides much less information than the library-generated message for an error. For instance, the result of **strerror(ERANGE)** is "math function value out of bounds," while the library message for this error includes the name and arguments of the failing function.

RETURN VALUE

The return value is a pointer to a message describing the error number.

EXAMPLE

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <lclib.h>

main()
{
   FILE *f;
   char *filename;

   quiet(1);    /* Suppress library messages. */

   f = fopen(filename, "w");

      /* Provide error message if open fails. */
   if (!f) printf ("Open failed. %s\n", strerror(errno));
}
```

RELATED FUNCTIONS

perror

SEE ALSO

"The errno Variable" on page 1-7

"System Macro Information" on page 1-8

"Diagnostic Control Functions" on page 2-11

strftime Convert Time to String

SYNOPSIS

```
#include <time.h>

size_t strftime(char *s, size_t maxsize, const char *format,
                const struct tm *timep);
```

DESCRIPTION

strftime converts a time value into a string according to the format specified by **format**. The string is placed in the array pointed to by **s**. No more than **maxsize** characters are placed into the array.

The format is a character sequence consisting of zero or more conversion specifications and regular characters. The conversion specifications are described below; ordinary characters, including the terminating-null character, are copied into the string without being converted.

The conversion specifications for **format** are as follows. Remember that the behavior of these specifications depend on the current locale. (See Chapter 10, "Localization," in the *SAS/C Library Reference, Volume 2* for more information on locales.) The "C" locale values for each of the specifications below are listed in Chapter 2, "Language Definition," in the *SAS/C Compiler and Library User's Guide.* **strftime** is affected by time zone information contained in the **TZ** environment variable, if it is defined.

%a	is replaced by the locale's abbreviated weekday name.
%A	is replaced by the locale's full weekday name.
%b	is replaced by the locale's abbreviated month name.
%B	is replaced by the locale's full month name.
%c	is replaced by the locale's date and time representation.
%d	is replaced by the day of the month as a decimal number from 01 to 31.
%H	is replaced by the hour as a decimal number from 00 to 23.
%I	is replaced by the hour as a decimal number from 01 to 12.
%j	is replaced by the day of the year as a decimal number from 001 to 366.
%m	is replaced by the month as a decimal number from 01 to 12.
%M	is replaced by the minute as a decimal number from 00 to 59.
%p	is replaced by the locale's equivalent of either a.m. or p.m.
%S	is replaced by the second as a decimal number from 00 to 59.
%U	is replaced by the week number of the year as a decimal number from 00 to 53, counting Sunday as the first day of the week.
%w	is replaced by the weekday as a decimal number from 0 to 6, with Sunday as 0.
%W	is replaced by the week number of the year as a decimal number from 00 to 53, counting Monday as the first day of the week.
%x	is replaced by the locale's date representation.
%X	is replaced by the locale's time representation.

strftime Convert Time to String
(continued)

%y	is replaced by the year without century as a decimal number from 00 to 99.
%Y	is replaced by the year with century as a decimal number.
%Z	is replaced by the time zone name or by no characters if a time zone cannot be determined.
%%	is replaced by %.

See Chapter 10, “Localization,” in the *SAS/C Library Reference, Volume 2* for a discussion of how locale affects the behavior of **strftime**. See Chapter 11, “Multibyte Character Functions,” in the *SAS/C Library Reference, Volume 2* for a discussion of the relationship between the format string for **strftime** and multibyte characters.

RETURN VALUE

If the conversion results in no more than **maxsize** characters, including the terminating-null character, **strftime** returns the number of resulting characters. This return value does not include the terminating-null character.

If the conversion results in more than **maxsize** characters, **strftime** returns 0. In this case, the contents of the array pointed to by **s** are indeterminate. The return value will be zero also if **strftime** is given an invalid format specifier, or if **strftime** fails for some reason other than the conversion resulting in more than **maxsize** characters.

CAUTION

If copying takes place between overlapping objects, the behavior of **strftime** is undefined.

If a conversion specification is not one of those listed above or some other error occurs while processing a specification, **strftime** issues a diagnostic, null terminates the conversion output array up to the specification that caused the error, and returns 0.

EXAMPLE

```
#include <time.h>
#include <stdio.h>

main()
{
   time_t now;
   struct tm *tm_ptr;
   char date_str[80];
   size_t nchar;

   time(&now);               /* Obtain today's date/time. */
   tm_ptr = localtime(&now); /* Convert value to a tm struct. */
```

strftime Convert Time to String
(continued)

```
      nchar = strftime(date_str, sizeof(date_str),
              "Today is %A, %B %d, %Y and %I:%M:%S %p is the time.",
               tm_ptr);
      printf("%.80s", date_str);
   }
```

RELATED FUNCTIONS

`asctime`, **`tzset`**

SEE ALSO

"Timing Functions" on page 2-11

strlen Compute Length of Null-Terminated String

SYNOPSIS

```
#include <string.h>

size_t strlen(const char *str);
```

DESCRIPTION

`strlen` returns the length of a null-terminated character string `str`.

RETURN VALUE

The return value is the length of the string, not counting the terminating null.

CAUTION

The scan for a null character continues until one is found, or until a protection or addressing exception occurs.

PORTABILITY

Note that many implementations before ANSI C define `strlen` to return `int` rather than `size_t`.

IMPLEMENTATION

If `<string.h>` is included (by an `#include` statement) and `strlen` is not undefined (by an `#undef` statement), `strlen` generates inline code. If the argument to `strlen` is a constant, the length is evaluated during compilation, and no code is generated for the function.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

#define MAXLINE 100

main()
{
   char line[MAXLINE+1];

   puts("Enter some text (at least 2 characters):");
   gets(line);
   puts("The last half of your text is:");
   puts(line + (strlen(line)/2));
}
```

RELATED FUNCTIONS

`mblen`

SEE ALSO

"String Utility Functions" on page 2-3

strlwr Convert a String from Uppercase to Lowercase

SYNOPSIS

```
#include <lcstring.h>

char *strlwr(char *str);
```

DESCRIPTION

strlwr converts uppercase alphabetic characters ('A' through 'Z') in the input string **str** into lowercase characters ('a' through 'z'). All other characters are unchanged.

strlwr is not affected by the program's locale.

RETURN VALUE

strlwr returns the original input string pointer **str**.

CAUTION

You must properly terminate the input string with the null character; otherwise, a protection or addressing exception may occur.

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int i;
   char *names[5];

   puts("Enter 5 last names, using only uppercase letters:");
   for( i = 0; i < 5; i++){
      names[i] = ( char *)malloc(256);
      printf("Enter name no.%d\n",i+1);
      gets(names[i]);
   }
      /* Convert each string in a table to lowercase letters. */
   puts("The names you have entered (converted to lowercase) "
        "are as follows:");
   for(i = 0; i < 5; i++)
      printf("%s\n",strlwr(names[i]));
}
```

RELATED FUNCTIONS

memlwr, **strupr**, **strxlt**

SEE ALSO

"String Utility Functions" on page 2-3

strncat Concatenate Two Null-Terminated Strings (Limited)

ISO ANSI UNIX

SYNOPSIS

```
#include <string.h>

char *strncat(char *to, const char *from, size_t maxlen);
```

DESCRIPTION

strncat copies characters from the second argument string (**from**) to the end of the first argument string (**to**) until a terminating-null character is found or until the number of characters specified by **maxlen** have been copied. After the maximum number of characters is reached, a terminating-null character is added to the output string.

RETURN VALUE

A pointer to the **to** string is returned.

CAUTION

A protection or addressing exception may occur if the **to** string is not properly terminated.

The effect of **strncat** is not defined if the **to** and **from** areas overlap.

If the **maxlen** value is 0, no characters are copied. If the value is negative, it is interpreted as a very large unsigned number, causing the number of characters copied to be essentially unlimited.

Because a null terminator is always appended to the **to** string, **maxlen+1** characters are copied if the length of the **from** string is greater than **maxlen**.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

#define MAXLINE 100
#define PRINTAMT 20

main()
{
   char line[MAXLINE];
   char intro[] = "Your input was:";
   char outline[sizeof(intro)+PRINTAMT]; /* space for output message */

   puts("Enter a line of input:");
   gets(line);
   strcpy(outline, intro);
   strncat(outline, line, PRINTAMT);     /* Append input to output.  */
   puts(outline);
   printf("Your input was truncated if it was longer"
          " than %d characters.", PRINTAMT);
}
```

strncat Concatenate Two Null-Terminated Strings (Limited)

(continued)

RELATED FUNCTIONS

`strcat`, **`strncpy`**

SEE ALSO

"String Utility Functions" on page 2-3

strncmp Compare Portions of Two Strings

SYNOPSIS

```
#include <string.h>

int strncmp(const char *str1, const char *str2,
            size_t maxlen);
```

DESCRIPTION

strncmp compares two character strings (**str1** and **str2**) using the standard EBCDIC collating sequence. The return value has the same relationship to 0 as **str1** has to **str2**. If 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.

RETURN VALUE

The return value from **strncmp** is 0 if the two strings are equal, less than 0 if **str1** compares less than **str2**, and 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 **strncmp**.

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** value. This may cause a protection or addressing exception, but this is unlikely because comparison ceases as soon as unequal characters are found.

IMPLEMENTATION

strncmp is implemented as a built-in function, unless you use it with **undef**.

EXAMPLE

Compare this example to the example for **strcmp**:

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   char command[20];
   int n = 0;

   do{
      n++;
       printf("You have executed this loop %d times.\n", n);
```

strncmp Compare Portions of Two Strings
(continued)

```
            puts("Enter quit(may be abbreviated) to end program,");
            puts(" or any other command to continue.");
            gets(command);
        }
        while(strncmp(command, "quit", strlen(command)) != 0);
        exit(0);
     }
```

RELATED FUNCTIONS

memcmp, **strcmp**

SEE ALSO

"String Utility Functions" on page 2-3

strncpy Copy a Limited Portion of a Null-Terminated String

SYNOPSIS

```
#include <string.h>

char *strncpy(char *to, const char *from, size_t maxlen);
```

DESCRIPTION

strncpy copies characters from the second argument string (**from**) to the first argument string (**to**) until a terminating-null character is found or until the number of characters specified by **maxlen** have been copied. If the maximum number of characters is reached, a terminating-null character is not added. If fewer than **maxlen** characters are copied, the **to** string is padded with enough null characters to bring the total number of characters copied to **maxlen**.

RETURN VALUE

The return value is a pointer to the **to** string.

CAUTION

If the **to** and **from** areas overlap, the effect of **strncpy** is not defined.

If the **maxlen** value is 0, no characters are copied. If the value is negative, it is interpreted as a very large unsigned number, probably causing massive overlay of memory.

Note: At the conclusion of a call to **strncpy**, the target string may not be null terminated.

PORTABILITY

Many implementations before ANSI C do not pad the target of **strncpy** with more than a single null.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

main()
{
   char *phrase = "It is almost 4:30 in the afternoon.";
   char after[11] ;
   char *colon;

   colon = strchr(phrase, ':');
   if (!colon)
      puts("No colon found in string.");
   else{
       strncpy(after, colon+1, 10); /* Copy up to 10 characters */
                                    /*  after a colon.         */
```

strncpy *(continued)* Copy a Limited Portion of a Null-Terminated String

```
            after[10] = '\0';
            printf("Text following colon is - %s\n", after);
                                        /* should print "30 in the ". */
         }
         return;
      }
```

RELATED FUNCTIONS

memcpy, **strncat**, **strcpy**

SEE ALSO

"String Utility Functions" on page 2-3

strpbrk Find First Occurrence of Character of Set in String

SYNOPSIS

```
#include <string.h>

char *strpbrk(const char *str, const char *cset);
```

DESCRIPTION

strpbrk locates the first occurrence in the first argument string (**str**) of a character in the second argument string (**cset**), returning a pointer to the character found.

RETURN VALUE

strpbrk returns a pointer to the requested character, or **NULL** if no character in the string is in the requested set of characters.

CAUTION

A protection or addressing exception may occur if either argument to **strpbrk** is not properly terminated.

See the **memscntb** function description for information on possible interactions between the **strpbrk** and **memscntb** or **strscntb** functions.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

main()
{
   char *line, *white, *temp;
   char input[80];

   puts("Enter some text:");
   line = gets(input);

      /* Locate the first white-space character in */
      /*  the line.                                */
   white = strpbrk(line, "\n\t\r\f\v ");
   puts("The first white space occurs after the word: ");

   for(temp = line;temp <= white;temp++)
   putchar(*temp);
   putchar('\n');
}
```

strpbrk Find First Occurrence of Character of Set in String
(continued)

RELATED FUNCTIONS

`strchr`, **`strcspn`**, **`strscan`**, **`strtok`**

SEE ALSO

"String Utility Functions" on page 2-3

strrchr Locate the Last Occurrence of a Character in a String

ISO ANSI UNIX

SYNOPSIS

```
#include <string.h>

char *strrchr(const char *str, int ch);
```

DESCRIPTION

strrchr searches an input string **str** for the last occurrence of a search character **ch**. The **strrchr** function is the reverse of **strchr**.

RETURN VALUE

strrchr returns a character pointer to the last occurrence of the search character in the input string, or **NULL** if the character is not found. If the search character is the null character ('\0'), the return value addresses the null character at the end of the input string.

CAUTION

A protection or addressing exception may occur if the input string is not properly terminated with the null character.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

#define MAXLINE 80

main()
{
   char text[MAXLINE];
   char *last_blank;

   puts("Enter some text.  Do not include trailing blanks.");
   gets(text);
   last_blank = strrchr(text, ' '); /* Find the last blank. */
   if (last_blank == NULL)
      puts("Your input was only a single word.");
   else if (*(last_blank+1) == '\0')
      puts("Your input included a trailing blank.");
   else
      printf("The last word in your input was \"%s\".",
             last_blank+1);
}
```

strrchr Locate the Last Occurrence of a Character in a String
(continued)

RELATED FUNCTIONS

`strchr`, **`strrcspn`**, **`strrspn`**

SEE ALSO

"String Utility Functions" on page 2-3

strrcspn Locate the Last Character in a Set

SYNOPSIS

```
#include <lcstring.h>

size_t strrcspn(const char *str, const char *cset);
```

DESCRIPTION

strrcspn scans the input string (**str**) for the last occurrence of any character in the search set (**cset**). The **strrcspn** function is the reverse of **strcspn**.

RETURN VALUE

strrcspn returns the number of characters in the argument string up to and including the last occurrence of any character in the search set.

If no character in the search set can be found in the input string, the value returned is 0. If the search set is null (that is, if it contains no characters), the return value from **strrcspn** is the length of the input string.

CAUTION

A protection or addressing exception may occur if either argument is not terminated with the null character.

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

void main()
{
   char *text, input[80];
   size_t len;
   int i;

   puts("Enter a line of text:");
   text = gets(input);

      /* Find the last blank space or punctuation */
      /*  character in a line terminated by '\0'. */
   len = strrcspn(text, ",.\"?;':!");

      /* Write to stdout all text after the last  */
      /*  punctuation character or space in the   */
      /*  string (which might not be a word).     */
   for(i = len; text[i] != '\0' && text[i] != '\n'; i++)
      putchar(text[i]);
   putchar('\n');
}
```

strrcspn Locate the Last Character in a Set
(continued)

RELATED FUNCTIONS

`strcspn`, **`strrchr`**, **`strrspn`**

SEE ALSO

"String Utility Functions" on page 2-3

strrspn Locate the Last Character of a Search Set Not in a Given Set

SYNOPSIS

```
#include <lcstring.h>

size_t strrspn(const char *str, const char *cset);
```

DESCRIPTION

strrspn locates the last character in the input string **str** not contained in the search set **cset**. The **strrspn** function is the reverse of **strspn**.

RETURN VALUE

strrspn returns the number of characters in the input string up to and including the last occurrence of a character not in the search set.

If all characters of the input string are in the search set, the return value is 0. If the search set is null (that is, if it contains no characters), the return value from **strrspn** is the length of the input string.

CAUTION

A protection or addressing exception may occur if either argument is not terminated with the null character.

EXAMPLE

This example uses **strrspn** to remove trailing blanks from the end of a line:

```
#include <lcstring.h>
#include <stdio.h>

#define MAXLINE 80

main()
{
   char *line;
   char string[MAXLINE];
   size_t i;

   puts("Enter a string, followed by some spaces:");
   line = gets(string);

      /* Find the position (i) of the last character   */
      /*  in the string that is not a blank.           */
   i = strrspn(line, " ");

      /* Check if line is a null string.               */
   if (i > 0)

         /* Set the character after the last           */
         /*  nonblank character to the null character. */
      line[i] = '\0';
```

strrspn *(continued)* Locate the Last Character of a Search Set Not in a Given Set

```
      else
         puts("The string contains only blanks.");

          printf("The last nonblank character in the string is %c\n",
                  line[i-1]);
   }
```

RELATED FUNCTIONS

strrcspn, **strspn**

SEE ALSO

"String Utility Functions" on page 2-3

strsave Allocate a Copy of a Character String

SYNOPSIS

```
#include <lcstring.h>

char *strsave(const char *str);
```

DESCRIPTION

`strsave` allocates a private copy of a character string (using the **`malloc`** function).

RETURN VALUE

`strsave` returns the address of the copy of the string, or **`NULL`** if no memory is available for a copy.

ERRORS

User ABEND 1205 and 1206 may occur if memory management data areas have been overlaid.

CAUTION

A protection or addressing exception may occur if the argument string is not properly terminated.

The copy should be released by a call to **`free`** when it is no longer required.

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

static char *filename;
static int couldopen(void);

main()
{
   if (couldopen() == 0)
      printf("The file \"%s\" was opened successfully.\n",
             filename);
   else
      printf("The file \"%s\" was not opened successfully.\n",
             filename);
   return;
}

int couldopen(void)
{
   char buf[FILENAME_MAX];
   FILE *f;
```

strsave Allocate a Copy of a Character String
(continued)

```
      puts("Enter a file name:");
      gets(buf);
      filename = strsave(buf);  /* Save the file name. Because */
                                /* buf is auto, saving the     */
                                /* address of buf is not safe. */
      f = fopen(buf, "r");
      if (f){
         fclose(f);
         return 0;              /* success                     */
      }
      else return -1;           /* failure                     */
   }
```

RELATED FUNCTIONS

strcpy

SEE ALSO

"String Utility Functions" on page 2-3

strscan Scan a String Using a Translate Table

SYNOPSIS

```
#include <lcstring.h>

char *strscan(const char *str, const char *table);
```

DESCRIPTION

strscan scans a null-terminated string (**str**) using a translate table (**table**). A *translate table* is an array of 256 bytes, one for each EBCDIC character. The scan terminates at the first character in the string whose table entry is not 0. The entry for the null character in the table (the first byte) should be nonzero to avoid scanning past the end of the string.

str is a pointer to the string to be scanned, and **table** addresses the first byte (corresponding to the character '\0') of the translate table. If any character in the string has a nonzero table entry, then the scan terminates at the first such character, and the address of that character is returned. If no character in the string before the terminating-null character has a nonzero table entry, the address of the null character is returned.

RETURN VALUE

strscan returns a pointer to the first character in the string whose table entry is nonzero, or the address of the terminating-null character if there is no such character and the table entry for '\0' is nonzero.

CAUTION

If the translate table does not contain a nonzero entry for the null character or if the **str** argument is not null terminated, **strscan** may search indefinitely for a character with a nonzero table entry. This may cause an 0C4 or 0C5 ABEND.

IMPLEMENTATION

strscan is implemented as a built-in function and uses the TRT instruction to search for a character with a nonzero entry in the table.

Tables generated by the **strscntb** function always define a nonzero table entry for the null character.

EXAMPLE

See the example for **strscntb**.

RELATED FUNCTIONS

memscan, **strscntb**

SEE ALSO

"String Utility Functions" on page 2-3

strscntb Build a Translate Table for Use by strscan

SYNOPSIS

```
#include <lcstring.h>

char *strscntb(char *area, const char *str, int val);
```

DESCRIPTION

strscntb builds a translate table that you use with the **strscan** function to scan a character string for the first occurrence of one of a set of characters. A translate table is an array containing 1 byte for each of the 256 EBCDIC characters. **strscan** scans its argument string for a character whose value in the translate table is not 0.

area is either the address of a 256-byte array, or **NULL**. If **area** is **NULL**, **strscntb** builds the translate table in a static area, whose address is returned. If **area** is not **NULL**, the table is built in the specified array.

str is a string of characters that are to be translated to the same value. The table byte corresponding to each character in **str** has the value specified by **val**, while all other characters have the value of **!val** (that is, 1 if **val** is 0, and 0 otherwise). For example, if **str** is "ab" and **val** is 0, bytes 129 and 130 (the EBCDIC decimal values for a and b) in the translate table have the value 0 and all other bytes (except for byte 0) have the value 1.

The table entry for the null character is a special case; this byte is always nonzero and forces **strscan** to terminate its scan at the end of the argument string.

When building the translate table with **strscntb**, you must consider how you will use **strscan**. If you are going to use **strscan** to search for the first occurrence of a character in **str**, **val** should be nonzero. If you want to search for the first character *not* in **str**, **val** should be 0.

RETURN VALUE

strscntb returns a pointer to the translate table. If **area** is **NULL**, this table may not be modified by the program.

CAUTION

If **strscntb** is called with a **NULL** area value, the table addressed by the return value is a static area. This area may be modified by the next call to any of these functions: **strscntb**, **memscntb**, **memchr**, **strchr**, **strcspn**, **strspn**, and **strpbrk**.

IMPLEMENTATION

strscntb is implemented as a built-in function. Inline code is generated if **str** is a string literal and **val** is an integer constant.

strscntb Build a Translate Table for Use by strscan
(continued)

EXAMPLE

This example using **strscntb** verifies that each string in an array contains only alphabetic characters and English punctuation. Using **strscan** rather than **strspn** is more efficient because a translate table only needs to be built once:

```
#include <lcstring.h>
#include <stdio.h>

#define SIZE 500

char *strings[SIZE];          /* string to be tested for */
                              /* unacceptable characters */
int i;
char engtable[256];           /* translate table         */

   /* Build table to skip letters and punctuation.        */
main()
{
   strscntb(engtable, "abcdefghijklmnopqrstuvwxyz"
                      "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
                      " ,./?:;'\ !-", 0);

   for (i = 0; i < SIZE; i++) {

         /* If unacceptable character is found in         */
         /*  string before null, print error message.     */
      if (*strscan(strings[i], engtable)) {
         printf("String %d contains unacceptable character:\n",
                i);
      }
   }
}
```

RELATED FUNCTIONS

memscntb, **strchr**, **strcspn**, **strscan**, **strspn**

SEE ALSO

"String Utility Functions" on page 2-3

strspn Locate the First Occurrence of the First Character Not in a Set

ISO ANSI UNIX

SYNOPSIS

```
#include <string.h>

size_t strspn(const char *str, const char *cset);
```

DESCRIPTION

strspn locates the first character in the argument string **str** not contained in the argument string **cset**, returning its position in **str**.

RETURN VALUE

strspn returns the number of consecutive characters in the character set **cset** found in the argument string **str**, starting at the first character in **str**. If all characters of the string are in the set (so that no character not in the set is found), the value returned is the length of the string. If the character set is null (that is, if it contains no characters), the return value from **strspn** is 0.

CAUTION

A protection or addressing exception may occur if either argument is not properly terminated.

See the **memscntb** function description for information on possible interactions between the **strspn**, **memscntb**, or **strscntb** functions.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

#define MAXLINE 40

main()
{
   char line[MAXLINE];
   char *word;
   size_t a,b;

   puts("Enter a word (only alphabetic characters):");
   word = gets(line);

      /* Find the position of the first character */
      /*  in the word that is not a vowel.       */
   a = strspn(word, "aeiou");

      /* Find the position of the first character */
      /*  in the word that is not a consonant.    */
   b = strspn(word, "bcdfghjklmnpqrstvwxyz");
```

strspn *(continued)* Locate the First Occurrence of the First Character Not in a Set

```
      printf("The first consonant in the given word is: %c\n",
              word[a]);
      printf("The first vowel in the given word is: %c\n",
              word[b]);
   }
```

RELATED FUNCTIONS

strcspn, **strrspn**, **strscan**

SEE ALSO

"String Utility Functions" on page 2-3

strstr Locate First Occurrence of a String within a String

SYNOPSIS

```
#include <string.h>

char *strstr(const char *str1, const char *str2);
```

DESCRIPTION

strstr scans the input string **str1** for the first occurrence of the search string **str2**.

RETURN VALUE

strstr returns a character pointer to the first occurrence of the search string in the input string. If the search string cannot be found, **strstr** returns **NULL**.

CAUTION

Both arguments must be terminated with the null character; otherwise, a protection or addressing exception can occur.

EXAMPLE

```
#include <string.h>
#include <stdio.h>

main()
{
   char *text_buffer ="12345-\n67-\n89";
   char *word_break ="-\n";
   char *hyphen;

   hyphen = strstr(text_buffer, word_break);
   printf("The first occurence of \"-\\n\" is after the digit %c\n",
          *(hyphen-1));
}
```

RELATED FUNCTIONS

strchr, **stcpm**

SEE ALSO

"String Utility Functions" on page 2-3

strtod Convert a String to Double

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *str, char **end);
```

DESCRIPTION

strtod expects 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.

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 returns the **double** value 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 370 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);
```

strtod Convert a String to Double
(continued)

```
      /* 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

`strtol`

SEE ALSO

"String Utility Functions" on page 2-3

strtok Get a Token from a String

SYNOPSIS

```
#include <string.h>

char *strtok(char *s1, const char *s2);
```

DESCRIPTION

strtok breaks the string pointed to by **s1** into a sequence of tokens, each of which is delimited by a character from the string pointed to by **s2**.

The tokens are created by a sequence of calls to **strtok**. In the first call in the sequence, **s1** points to the string to be broken down. In subsequent calls in the sequence, **s1** is **NULL**. The string pointed to by **s2** may be different from call to call.

If **s1** is not **NULL** (that is, the call is the first call in a sequence), then **strtok** searches the string pointed to by **s1** for the first character that is not contained in the string pointed to by **s2**. If such a character is found, then it is the start of the first token. If no such character is found, **strtok** returns **NULL**. Otherwise, the character that was found becomes the start of the first token.

After a token has been started, **strtok** searches for the first character contained in the string pointed to by **s2**. If such a character is found, it is replaced by **strtok** with a null character, thereby terminating the token. If no such character is found, the token extends to the null character terminating the string pointed to by **s1**. In either case, **strtok** returns the start of the token.

Subsequent calls to **strtok** behave as described in the previous paragraph, starting at the character following the null character terminating the previous token. However, if the previous token extended to the end of the string pointed to by **s1**, **strtok** simply returns **NULL**.

RETURN VALUE

strtok returns a pointer to the start of a token, or **NULL** if there is no token.

CAUTION

The pointer that is saved by **strtok** is kept in an **extern** variable that is local to the calling load module. Therefore, in a multiload module program, a sequence of calls to **strtok** for a given string must be made from the same load module.

strtok Get a Token from a String
(continued)

EXAMPLE

This example using **strtok** breaks out words separated by blanks or commas:

```
#include <string.h>
#include <stdio.h>
#include <stddef.h>

main()
{
   char test[] = "first,second,third,fourth";
   char* token;

   token = strtok(test, ", ");
   while(token != NULL){
      puts(token);

         /* Continue scan from where it left off. */
      token = strtok(NULL, ", ");
    }
}
```

RELATED FUNCTIONS

strchr, **strcspn**, **strspn**

SEE ALSO

"String Utility Functions" on page 2-3

strtol Convert a String to Long Integer

SYNOPSIS

```
#include <stdlib.h>

long int strtol(const char *str, char **end, int radix);
```

DESCRIPTION

strtol converts a character string to a **long** integer. The string is expected to contain the representation of an integer in base **radix**. Base **radix** can contain an integer between 2 and 36; if it is larger than 10, letters 'a' through 'z' (either case) are interpreted as digits greater than 10. If **radix** is 16, a leading 0x may be present in the string, but it is ignored. Initial white space characters are always ignored.

If **radix** is 0, the base is determined by the initial characters of the string (after leading white space and an optional sign). That is, if the string begins with 0x or 0X, the base is assumed to be 16; if it begins with 0, it is assumed to be 8; otherwise, it is assumed to be 10.

If the **end** value is not **NULL**, ***end** is modified to address the first character of the string that is not a valid base **radix** digit. However, if no initial segment of the string can be interpreted as an integer of appropriate base, **str** is assigned to ***end**.

RETURN VALUE

strtol returns the integer value represented by the character string, up to the first unrecognized character. If no initial segment of the string can be interpreted as an integer of appropriate base, **0L** is returned.

DIAGNOSTICS

If the correct value is too large to be stored in a 370 **long**, **errno** is set to ERANGE and either **LONG_MAX** ($2^{31}-1$) or **LONG_MIN** (-2^{31}) is returned, depending on the sign of the value.

EXAMPLE

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>

long int hextol(char *);
static int failed;

main()
{
   char *string;
   char input[20];
   long output;
```

strtol Convert a String to Long Integer
(continued)

```
   puts("Enter a hex string to convert to long int:");
   string = gets(input);
   output = hextol(string);
   if (!failed)
      printf("The value of the string, printed in decimal, is: %ld\n",
              output);
}

long int hextol(char *hexstr)
{
   long value;
   char *stopchar;    /* where strtol conversion stops */

      /* Skip space characters.                        */
   while(isspace(*hexstr)) ++hexstr;

      /* Skip leading 0x.                              */
   if (*hexstr == '0' && tolower(*(hexstr+1)) == 'x')
      hexstr += 2;

      /* refused signed hex value                      */
   if (*hexstr == '+' || *hexstr == '-') {
      puts("Unsigned hex only please");
      failed = 1;
      return -1L;
   }

      /* Convert hex to long.                          */
   value = strtol(hexstr, &stopchar, 16);

      /* Determine whether string is valid.            */
   if (stopchar == hexstr)  {
      printf("Invalid hex string: %s\n", hexstr);
      failed = 1;
   }

      /* Check for characters after digits.            */
   else if (*stopchar && !isspace(*stopchar))
      printf("Extra characters after hex value ignored: %s\n",
              stopchar);

   return value;
}
```

RELATED FUNCTIONS

strtod, **strtoul**

SEE ALSO

"String Utility Functions" on page 2-3

strtoul Convert a String to an Unsigned Long Integer

SYNOPSIS

```
#include <stdlib.h>

unsigned long int strtoul(const char *str, char **end, int base);
```

DESCRIPTION

strtoul converts a character string to an **unsigned long** integer. The string is expected to contain the representation of an unsigned integer in base **radix**. Base **radix** may contain an integer between 2 and 36; if it is larger than 10, letters 'a' through 'z' (either case) are interpreted as digits greater than 10. If **radix** is 16, a leading 0x may be present in the string, but it is ignored. Initial white space characters are always ignored.

If **radix** is 0, the base is determined by the first character of the string (after leading white space and an optional sign). That is, if the string begins with 0x or 0X, the base is assumed to be 16; if it begins with 0, it is assumed to be 8; otherwise, it is assumed to be 10.

If the **end** value is not 0, ***end** is modified to address the first character of the string that is not a valid base **radix** digit. However, if no initial segment of the string can be interpreted as an integer of appropriate base, **str** is assigned to ***end**.

RETURN VALUE

strtoul returns the unsigned integer value represented by the character string up to the first unrecognized character. If no initial segment of the string can be interpreted as an integer of appropriate base, **0UL** is returned.

DIAGNOSTICS

If the correct value is too large to be stored in a 370 **unsigned long**, **errno** is set to ERANGE, and $2^{32}-1$ is returned. (This is the value of **ULONG_MAX**, defined in **<limits.h>**.)

EXAMPLE

```
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>

unsigned long hextoul(char *);
static int failed;

main()
{
   char *string;
   char input[20];
   unsigned long output;
```

strtoul Convert a String to an Unsigned Long Integer
(continued)

```
      puts("Enter a hex string to convert to unsigned long int:");
      string = gets(input);
      output = hextoul(string);
      if (!failed)
         printf("The string is converted to: %ld\n", output);
   }

   unsigned long hextoul(char * a)
   {
      unsigned long value;
      char *hexstr;     /* hexadecimal input string       */
      char *stopchar;   /* where strtoul conversion stops */
      hexstr = a;

         /* Skip space characters.                        */
      while(isspace(*hexstr)) ++hexstr;

         /* Skip leading 0x.                              */
      if (*hexstr == '0' && tolower(*(hexstr+1)) == 'x')
         hexstr += 2;

         /* refused signed hex value                      */
      if (*hexstr == '+' || *hexstr == '-') {
         puts("Unsigned hex only please");
         failed = 1;
         return -1UL;
      }
         /* Convert hex to long.                          */
      value = strtoul(hexstr, &stopchar, 16);

         /* Determine whether string is valid.            */
      if (stopchar == hexstr) {
         printf("Invalid hex string: %s\n", hexstr);
         failed = 1;
      }
         /* Check for characters after digits.            */
      else if (*stopchar && !isspace(*stopchar))
         printf("Extra characters after hex value ignored: %s\n",
                stopchar);

      return value;
   }
```

RELATED FUNCTIONS

strtol

SEE ALSO

"String Utility Functions" on page 2-3

strupr Convert a String from Lowercase to Uppercase

SYNOPSIS

SAS/C

```
#include <lcstring.h>

char *strupr(char *str);
```

DESCRIPTION

strupr converts lowercase alphabetic characters ('a' through 'z') in the input string **str** into uppercase characters ('A' through 'Z').

strupr is not affected by the program's locale.

RETURN VALUE

strupr returns a pointer to the original input string.

CAUTION

The input string must be properly terminated with the null character; otherwise, a protection or addressing exception can occur.

EXAMPLE

```
#include <lcstring.h>
#include <stdio.h>

#define MAXLEN 80

main()
{
   char line[MAXLEN];
   char *input;

   puts("Enter a string of lowercase characters:");
   input = gets(line);
   strupr(input);
   printf("Your converted input is as follows:\n%s\n",input);
}
```

RELATED FUNCTIONS

memupr, **strlwr**, **strxlt**

SEE ALSO

"String Utility Functions" on page 2-3

strxlt Translate a Character String

SYNOPSIS

```
#include <lcstring.h>

char *strxlt(char *str, const char *table);
```

DESCRIPTION

`strxlt` translates a null-terminated string from one character set to another. The first argument is the address of the string (**`str`**) to be translated. **`table`** is a pointer to a 256-byte translation table, which should be defined so that **`table[c]`** for any character **`c`** is the value to which **`c`** should be translated. (The function **`xltable`** can frequently be used to build such a table.)

The argument string is translated in place; that is, each character in the string is replaced by a translated character. The null character that terminates the string is never translated.

RETURN VALUE

The return value is a pointer to the translated string.

CAUTION

If the source string and the translation table overlap, the effect of **`strxlt`** is not defined.

IMPLEMENTATION

`strxlt` is implemented by inline code unless the function is undefined (by an **`#undef`** statement) to prevent this.

EXAMPLE

```
#include <lcstring.h>

char punctab[256] ;
char *number, *where;

   /* Build a table to interchange comma and period. */
xltable(punctab, ",.", ".,");
.
.
.

   /* Interchange comma and period for European      */
   /*  conventions.                                  */
if (strcmp(where, "Europe") == 0)
   strxlt(number, punctab);
```

strxlt Translate a Character String
(continued)

RELATED FUNCTIONS

`memxlt`, **`strlwr`**, **`strupr`**, **`xltable`**

SEE ALSO

"String Utility Functions" on page 2-3

symlink Make Symbolic Link

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int symlink(const char *path, const char *sl);
```

DESCRIPTION

symlink creates a symbolic link to an OpenEdition HFS file. The file need not already exist. **path** is the pathname of the file. **sl** is the pathname to be assigned to the symbolic link.

When you call **symlink** in a non-**posix** -compiled application, the pathname and the link name both are interpreted according to the normal rules for interpretation of filenames. These names should include a style prefix if the default style is not **''hfs'')**. Note that the style prefix is not actually stored in the symbolic link.

RETURN VALUE

symlink returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

This example attempts to define the second argument as a hard link to the first argument. If this fails with **errno** equal to **EXDEV**, indicating that links are not supported between file systems, the second argument is created as a symbolic link instead.

```
/* This example must be compiled using the posix compiler option. */

#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[]) {
   int rc;
   if (argc != 3) {
      fputs("Incorrect number of arguments.", stderr);
      exit(EXIT_FAILURE);
   }
   rc = link(argv[1], argv[2]);  /* Try to make hard link. */
   if (rc != 0)                  /* if link failed         */
      if (errno != EXDEV) {      /* unexpected error       */
         perror("link error");
         exit(EXIT_FAILURE);
      } else {                   /* Attempt symbolic link. */
         rc = symlink(argv[1], argv[2]);
         if (rc != 0) {
            perror("symlink error");
            exit(EXIT_FAILURE);
         }
```

symlink Make Symbolic Link
(continued)

```
            printf("%s was created as a symbolic link to %s.\n",
                   argv[2], argv[1]);
         }
      else printf("%s was created as a hard link to %s.\n",
                  argv[2], argv[1]);
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

`link`, **`lstat`**, **`readlink`**

SEE ALSO

Chapter 19, “Introduction to POSIX,” in *SAS/C Library Reference, Volume 2*

“File Management Functions” on page 2-16

system Execute a System Command

POSIX ISO ANSI UNIX

SYNOPSIS

```
#include <stdlib.h>

int system(const char *cmd);
```

DESCRIPTION

The **system** function executes a character string as a command by passing it to an operating-system-defined command processor. The **cmd** argument is a pointer to the command string, which consists of two parts: a prefix that contains an optional command type followed by a colon and a command.

Note: The case of the command name or its arguments or both may be significant, depending on the operating system and the command being executed.

RETURN VALUES

The **system** function returns an integer status code indicating the success or failure of the command. The status code for a successful command is normally 0. If called with a **NULL** argument, **system** returns 1 to indicate that command processing is available.

In the 370 implementation, the status code is normally the return code (or completion code) of the invoked program or command, unless an error is detected by the **system** function itself.

Certain status codes returned by **system** have special significance. These codes are given symbolic names by the header file **<lclib.h>**. Here are the names of these codes and their meanings:

SYS_ATTN	indicates the command was terminated by attention.
SYS_ABTM	indicates the command was abnormally terminated.
SYS_CUNK	indicates an unknown command.
SYS_CSYN	indicates a command syntax error.
SYS_INF	indicates a system interface failure.
SYS_TNAC	indicates the specified command environment was not active.
SYS_TSYN	indicates a command-type syntax error.
SYS_TUNK	indicates an unknown command type.
SYS_CHILD	indicates a problem creating an OpenEdition child process.

Some of these codes are meaningful only for one operating system, and the situations in which they are returned are dependent on the operating system.

CAUTION

If the **TSO:** prefix is used with the **system** function, unpredictable results may occur under MVS/XA when running in 31-bit addressing mode and TSO/E is not installed. (In this case, TSO commands cannot be invoked with the **system** function.)

Use **system** with the **TSO:** prefix in 31-bit mode programs only if TSO/E is installed.

system Execute a System Command
(continued)

DIAGNOSTICS

Errors in the command string can generate library messages, operating-system messages, or both.

PORTABILITY

The format of the command string and the meaning of the status code for **system** is completely system dependent, and it is unlikely that a program calling **system** can be moved to another system without modification.

system is a POSIX.2 function.

IMPLEMENTATION

MVS

Under MVS, **system** can use a prefix of **PGM:**, **TSO:** or **SH:**. If the prefix is omitted, **SH:** is assumed if the program was compiled with the **posix** option; otherwise, **PGM:** is assumed. For programs compiled with the **posix** option, the argument to **system** is always assumed to be a shell command, even if it appears to have an explicit prefix. To use the **PGM:** or **TSO:** prefix in a **posix**-compiled program, you must precede the prefix with **``//''**, for example, **system(``//PGM:IEFBR14'')**. The **``//''** prefix is recognized whether or not a program is compiled with the **posix** option to enable you to write subroutines that call **system** and can be used in both **posix**-compiled and non-**posix**-compiled programs. The effect of using the **PGM:** and **TSO:** prefixes follows:

- If the prefix **PGM:** is specified or defaulted, the first token of the command string is interpreted as the name of a load module and the remainder as a PARM string. For example, the following statement calls the load module IEBUPDTE, passing the PARM string **``NEW''**:

  ```
  rc = system("PGM: IEBUPDTE NEW");
  ```

  To put it another way, IEBUPDTE is called in the same way that it is called by the following JCL statement:

  ```
  // EXEC PGM=IEBUPDTE,PARM='NEW'
  ```

  The PARM string always begins with the character after the separator following the program name.
- When you use the **TSO:** prefix, the first token in the command string is invoked as a TSO command, with the rest of the string as its arguments. The **system** function can be used to invoke TSO CLISTs or REXX EXECs, provided that the CLIST ATTN statement is not used. (If this statement is used and an attention interrupt occurs, the results are unpredictable.)

It is recommended that a program that calls **system** using the **TSO:** prefix be executed as a TSO command. When a C program calls **system** to execute a TSO command, the library must locate TSO interface information, which is readily available to programs that have been called as commands. For programs that are not called as commands, this information must be extracted from unprotected system-control blocks. Because these control blocks are unprotected, it is possible for a malfunctioning TSO program (either the C program or one executed earlier) to overlay this information. Any attempt to

system Execute a System Command
(continued)

execute a TSO command using the corrupted control blocks may result in a program ABEND, involuntary logoff, or incorrect results.

Under TSO, if an attention interrupt occurs during a call to `system`, the called program or command is immediately terminated.

When running with TSO/E Release 1.3 or higher under MVS/XA or MVS/ESA, `system` uses the TSO service routine IKJEFTSR to invoke TSO commands, CLISTs, or EXECs. This interface enables an unauthorized program to call commands that require authorization.

The IKJEFTSR interface is sensitive to TSO release and maintenance levels, and it can behave differently from release to release. You should be aware of these points:

- For TSO/E Release 2 or greater, no message is produced when an attempt is made to execute a CLIST that cannot be located. The error is indicated by a return code of SYS_CUNK from the system. For earlier releases, the operating system generates the message `COMMAND NOT FOUND` in this situation, and `system` returns a code of 12.
- If you use `system` to invoke the TSO CALL command, and any subtask of the called program terminates abnormally, all tasks of the called program are immediately terminated, and `system` returns with a code of SYS_ABTM.
- For Versions 1.3 and 1.4 of TSO/E, IKJEFTSR creates a parallel terminal monitor program (TMP) to execute commands. For this reason, TASKLIBs defined for the executing C program are not available to the parallel command. If you are executing the CALL command, only the library specified by the CALL command is used as a TASKLIB. For Version 2 of TSO/E, this does not apply unless either the calling or the called program is authorized. For more information on IKJEFTSR, refer to the IBM publication, *TSO Extensions Programming Services*, (SC28-1875).
- Do not use the `system` function to invoke C programs that use ISPF services. The attention handling of ISPF overrides the SAS/C library's attention handling, and you may be unable to use an attention interrupt to terminate the called program. ISPF applications should be invoked by the ISPF SELECT service.

OpenEdition

In addition to its use for invoking MVS load modules and TSO commands, you can use the `system` function to invoke shell commands if OpenEdition is installed and available.

Note: If a program uses any OpenEdition features, such as HFS files or POSIX signals, you should not use the `system` function prefixes `PGM:` or `TSO:` to invoke another program that also uses OpenEdition. OpenEdition will treat both programs as comprising a single process, which can cause confusing behavior in file access, signal handling, and other areas. In general, use the `system` function with the `SH:` prefix or the `oeattach` function to invoke one OpenEdition application from another.

- For programs that are not compiled with the `posix` option, invocation of a shell command is requested by prepending the command name with the prefix `''SH:''`. For instance, for a non-POSIX-compiled program, the call `system(''SH:ps -e'')` invokes the shell command `''ps''` with the argument string `''-e''`.

system Execute a System Command
(continued)

- For programs that are compiled with the **posix** option, all calls to **system** are treated as **''SH:''** style, unless the argument string begins with the escape characters **''//''**. Thus, for programs compiled with the **posix** option, the call **system(''ps -e'')** invokes the shell command **ps**, while the call **system(''//pgm:iefbr14'')** invokes the MVS load module IEFBR14.
- As with filenames, the **''//''** prefix is recognized whether or not a program is compiled with **posix**, so that the form **system(''//SH:ps -e'')** can be used to call a shell command regardless of the compilation mode of the program.
- When the **system** function is called to invoke a shell command, it does so by forking a child, which then calls one of the **exec** programs from the shell to run the command. This is in accordance with the POSIX 1003.2 definition of the **system** function. If the program catches the SIGCHLD signal, an instance of this signal is raised as a result of the termination of the shell invoked by the **system** function; therefore, the program must be prepared to deal appropriately with the signal. Note that the **system** function temporarily ignores the POSIX SIGINT and SIGQUIT signals when it invokes the shell, as required by the 1003.2 draft standard.
- When **system** is called to invoke a shell command, it invokes the file **/bin/sh**, which is normally the OpenEdition shell. If the OpenEdition shell is not installed, **/bin/sh** may be some other shell, or it may not exist. If **/bin/sh** does not exist, **system** invokes the program defined by the user's **uid** definition as the initial user program. If this program (or **/bin/sh**) is not a POSIX-conforming shell program, the behavior of a POSIX-conforming program that uses **system** may deviate from the standard.

If the **system** function is successful at calling a shell command, its return value is the exit status code of the shell, which can be interpreted by the **<wait.h>** macros such as **WEXITSTATUS** and **WTERMSIG**. If no child process can be created, **system** returns −1, which is given the symbolic name **SYS_CHLD** in **<lclib.h>**.

CMS

Under CMS, **system** can use a prefix of **CMS:**, **CP:**, **SUBSET:**, or **XEDIT:**. If the prefix is omitted, **CMS:** is assumed. The prefix can also be preceded by **''//''** for MVS compatibility. These paragraphs describe the effects of each of these prefixes:

- The **CMS:** prefix has no effect, and the command string is executed as if it were entered from the command line.

  system issues the command with the CMS command-search function, which uses the entire CMS command-search hierarchy. (See the *VM/ESA CMS User's Guide* for more information.) For example, the IMPEX and IMPCP settings are respected. The command string does not have to be uppercase when you use the **CMS:** prefix.

  The **system** function always passes both tokenized and untokenized parameter lists to the command. Be careful not to use **system** to invoke a program that runs in the user area if the calling program is already running in the user area.

system Execute a System Command
(continued)

- □ If you use the **CP:** prefix, **system** assumes the command is a CP command and assures that CP is called explicitly.
  system uppercases the command and issues it with DIAGNOSE X'08'. In this example, the calls to **system** are not the same:

```
rc = system("CP Q DASD");
rc = system("CP:Q DASD");
```

  The first call invokes the CMS command CP and the second call invokes DIAGNOSE X'08'.
- □ A command with the **SUBSET:** prefix is not executed unless it is a CMS SUBSET command. Commands other than SUBSET cause **system** to return **SYS_CUNK**. Using the **SUBSET:** prefix prevents CMS user area commands from overlaying the C program if the C program is already running in the user area. For more information about the CMS SUBSET command, refer to the appropriate IBM publication.
- □ If you use the **XEDIT:** prefix is used, **system** issues the command as an XEDIT subcommand. If XEDIT is not active, **system** returns SYS_TNAC.

If the command prefix is unknown (that is, it is not one of the prefixes listed here), it is treated as the name of a subcommand environment. A *subcommand environment* is a program that has been named with the CMS SUBCOM function. If the subcommand environment is active, the command is transferred (with the CMS command search function) to the subcommand environment. If the subcommand environment is not active, **system** returns **SYS_TNAC**. For more information about subcommand environments, refer to the appropriate IBM publication.

EXAMPLE

This example creates a new PDS named EXAMPLE.OUTPUT and writes member README. If the PDS already exists, it is not changed.

```
#include <stdlib.h>
#include <stdio.h>

main()
{
   int sysrc;
   FILE *readme;

   puts("Calling the TSO allocate command to create EXAMPLE.OUTPUT");
   sysrc = system("tso: allocate da(example.output) new sp(1 1) tr "
                  "dir(1)");       /* Allocate example.output NEW. */
   if (sysrc != 0){
         puts("Unable to allocate EXAMPLE.OUTPUT. File probably "
              "already exists.");
         exit(EXIT_FAILURE);
   }
```

system Execute a System Command
(continued)

```
      readme = fopen("tso:example.output(readme)", "w");
      fputs("This file was created by a SAS/C example program.\n",
             readme);
      fclose(readme);
      exit(EXIT_SUCCESS);
   }
```

RELATED FUNCTIONS

`fork`, **`oslink`**, **`popen`**

SEE ALSO

“System Interface and Environment Variables” on page 2-17

Chapter 19, “Introduction to POSIX” in *SAS/C Library Reference, Volume 2*

tan Compute the Trigonometric Tangent

SYNOPSIS

```
#include <math.h>

double tan(double x);
```

DESCRIPTION

tan computes the trigonometric tangent of an argument **x** expressed in radians.

Because the tangent function is periodic, only the value of x mod 2π is used to compute the tangent. If **x** is very large, only a limited precision is left to represent x mod 2π. Thus, an error message is written for very large negative or positive arguments (see DIAGNOSTICS).

RETURN VALUE

tan returns the value of the tangent of the argument **x**, provided that this value is defined and computable. The return value is a double-precision, floating-point number.

DIAGNOSTICS

The tangent is not defined if **x** is $\pm\pi/2$, $\pm 3\pi/2$, or any other value of the following form:

$$\pi/2 + n\pi$$

n is an integer.

If the value of **x** is so close to an odd multiple of $\pi/2$ that the tangent cannot be represented accurately, the function returns **HUGE_VAL**. The run-time library writes an error message to **stderr** (the standard error file).

If the value of **x** is greater than 6.7465e9, the function returns 0.0. In this case, the message "total loss of significance" is also written to **stderr**.

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

EXAMPLE

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

#define YVAL 1.04

main()
{
   double cotan;

      /* The cotangent is 1 divided by the */
      /*  tangent of YVAL.                 */
   cotan = 1 / tan(YVAL);
   printf("1 / tan(%f) = %f\n", YVAL, cotan);
}
```

tan Compute the Trigonometric Tangent
(continued)

RELATED FUNCTIONS

`cos`, `_matherr`, `sin`

SEE ALSO

"Mathematical Functions" on page 2-6

tanh Compute the Hyperbolic Tangent

SYNOPSIS

```
#include <math.h>

double tanh(double x);
```

DESCRIPTION

tanh computes the hyperbolic tangent of its argument x, as expressed by this relation:

$$r = (e^{x} - e^{-x}) / (e^{x} + e^{-x})$$

RETURN VALUE

tanh returns the principal value of the hyperbolic tangent of the argument **x**, provided that this value is defined and computable. The return value is a double-precision, floating-point number in the closed interval [−1.0,1.0].

DIAGNOSTICS

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

EXAMPLE

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

#define INVAL 3.8

main()
{
   double hyper_tan;
   hyper_tan = tanh(INVAL);
   printf("tanh(%f) = %f\n", INVAL, hyper_tan);
}
```

RELATED FUNCTIONS

cosh, **_matherr**, **sinh**

SEE ALSO

"Mathematical Functions" on page 2-6

time Return the Current Time

SYNOPSIS

```
#include <time.h>

time_t time(time_t *timep);
```

DESCRIPTION

`time` returns the number of seconds from the start of an implementation-defined era. If the **`timep`** pointer is not **`NULL`**, this value is also stored in the storage addressed by **`timep`**. The type of **`time`** and of the data referenced by its argument is **`time_t`**, declared in the header file **`<time.h>`**. This is a numeric type (implemented on the IBM 370 system as **`double`**).

RETURN VALUE

`time` returns the approximate number of seconds since the start of the epoch. The 1970 default epoch starts at midnight GMT, Jan. 1, 1970, as required by the POSIX.2 standard. See "Timing Functions" on page 2-11 for information on defining a different epoch.

DIAGNOSTICS

`(time_t)-1` is returned if the time cannot be determined.

PORTABILITY

SAS/C defines the type **`time_t`** as double. Because most C implementations define **`time_t`** as a long integer, some applications assume this equivalence. Such applications will require modifications for use with SAS/C.

IMPLEMENTATION

`time` returns the contents of the 370 time-of-day clock after conversion to **`time_t`** format and adjustment for the epoch.

EXAMPLE

```
#include <time.h>

time_t before, after;
main()
{
   time(&before);          /* Get time before computation. */
   compute();
   time(&after);           /* Get time after computation.  */
   printf("Elapsed time for computation = %10.4f seconds\n",
           difftime(after, before));
}
```

time Return the Current Time
(continued)

RELATED FUNCTIONS

`difftime`

SEE ALSO

"Timing Functions" on page 2-11

tmpfile Create and Open a Temporary File

SYNOPSIS

```
#include <stdio.h>

FILE *tmpfile(void);
```

DESCRIPTION

tmpfile creates a temporary file. You can open the file for both reading and writing. When you close the file, it is deleted. Note that the meanings of the words create and temporary are system dependent.

For programs compiled with the **posix** option, the temporary file created by **tmpfile** is an HFS file in the directory **/tmp**. For non-**posix**-compiled programs, the file is an MVS or CMS disk file suitable for **''rel''** access. For MVS non-**posix**-compiled programs, a file created by **tmpfile** is a true temporary file, and is deleted at the end of the job step or TSO session, even if the program terminates abnormally. For **posix**-compiled programs or programs on CMS, a file created by **tmpfile** is not deleted if the calling program is terminated abnormally.

RETURN VALUE

tmpfile returns a pointer to the **FILE** object associated with the temporary file (or **NULL** if the temporary file cannot be created).

PORTABILITY

Temporary files, either created through the **tmpfile** function or through the UNIX style I/O functions, vary with the operating system. However, **tmpfile** is portable unless a program depends on special properties of **''rel''** files.

EXAMPLE

See the example for **fscanf**.

RELATED FUNCTIONS

tmpnam

SEE ALSO

"Temporary files under MVS" on page 3-19

"Temporary files under CMS" on page 3-20

"I/O Functions" on page 2-13

tmpnam Generate Temporary Filename

SYNOPSIS

```
#include <stdio.h>

char *tmpnam(char *buf);
```

DESCRIPTION

tmpnam generates a string that is a valid filename and is not the same as the name of any existing file. If a file with this name is opened, it continues to exist after program termination. **tmpnam** generates up to **TMP_MAX** filenames, a different name each time it is called.

For a program compiled with the **posix** option, the name returned by **tmpnam** defines a file in the HFS directory **/tmp**.

RETURN VALUE

If **buf** is **NULL**, **tmpnam** leaves its result in an internal static object and returns a pointer to that object. Subsequent calls to **tmpnam** may modify that same object.

If **buf** is not **NULL**, it is assumed to point to an array of at least **L_tmpnam** characters; **tmpnam** writes its result in that array and returns the argument as its value.

CAUTION

The generated filenames are designed to be unique. The library makes 100 attempts to generate a unique filename at each call to **tmpnam**. If after 100 attempts it cannot generate a unique filename, **tmpnam** returns **NULL**.

IMPLEMENTATION

This implementation essentially assigns a value of infinity to **TMP_MAX** because it is virtually impossible to cause an error by calling **tmpnam** too many times.

The returned filename strings are composed as follows:

MVS

```
dsn:userid.jobid. $ddmonyr.$hhmmss. $tens-of-microseconds
```

Here is an example:

```
dsn:GEORGE.JOB01234.$10NOV88.$142253.$0000792
```

If a userid is not available, use **''C-TMP''**. Here is an example:

```
dsn:C-TMP.JOB01234.$10NOV88.$142253.$0000792
```

CMS

```
cms:$ddmonyr  $tens-of-microseconds fml
```

Here is an example:

```
cms:$10NOV88 $0000792 A1
```

tmpnam Generate Temporary Filename
(continued)

The CMS filemode letter is chosen from the read and write disk with the most space.

OpenEdition

```
/tmp/logonid.pid.ddmonyr.hhmmss.tens-of-microseconds
```

For example:

```
/tmp/JANE.524290.10NOV94.142253.0000792
```

EXAMPLE

```
#include <stdio.h>
#include <stdlib.h>

main()
{
   char *name;
   FILE *temp;
   name = tmpnam(NULL);
   if (name == NULL) exit(EXIT_FAILURE);
   temp = fopen(name, "w");
   fputs("Hello, temporary world.\n", temp);
   fclose(temp);
   remove(name);
}
```

RELATED FUNCTIONS

`tmpfile`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1
“I/O Functions” on page 2-13

toebcdic Reduce Integer to EBCDIC Character

SYNOPSIS

```
#include <lctype.h>

int toebcdic(int i);
```

DESCRIPTION

toebcdic reduces an integer **i** to an EBCDIC character by turning off all bits not stored in a **char** value.

toebcdic corresponds to the UNIX C compiler function **toascii**, which is not meaningful except when ASCII is the native character set.

RETURN VALUE

toebcdic returns the corresponding EBCDIC character value.

EXAMPLE

```
#include <lctype.h>
#include <stdio.h>

main()
{
int i,input;

   for(;;) {
      puts("Enter an integer (0 to quit)");
      scanf("%d", &input);
      if (feof(stdin) || input == 0) break;
      i = toebcdic(input);
      if (isprint(i))
         printf("The EBCDIC character for the integer %d is '%c'.\n",
                input, i);
      else
         printf("The EBCDIC character for the integer %d is "
                "('\\x%.2x' not printable).\n", input, i);
   }
}
```

RELATED FUNCTIONS

isebcdic

SEE ALSO

"Character Type Macros and Functions" on page 2-1

tolower Translate Uppercase Character to Lowercase

SYNOPSIS

```
#include <ctype.h>

int tolower(int c);
```

DESCRIPTION

`tolower` translates an uppercase character to the corresponding lowercase character. The argument must either be a **`char`** value or **`EOF`**. The mapping of uppercase to lowercase characters is locale dependent.

RETURN VALUE

If the argument is an uppercase character, the corresponding lowercase character is returned; otherwise, the argument value is returned.

IMPLEMENTATION

`tolower` is implemented by the compiler as a built-in function, unless you use the name **`tolower`** with **`#undef`**.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define MAX 40

main()
{
   char *str, *ptr;
   char input[MAX];

   puts("Enter a string of uppercase characters (maximum of 40):");
   str = gets(input);
   ptr = str;

      /* Translate all uppercase characters in a string to      */
      /*  lowercase characters.                                 */
   while (*str) {
      *str = tolower(*str);

         /* Increment outside of macro for maximum portability. */
      str++;
   }
   printf("%s\n", ptr);
}
```

tolower Translate Uppercase Character to Lowercase
(continued)

RELATED FUNCTIONS

`islower`, `memlwr`, `strlwr`, `toupper`

SEE ALSO

Chapter 10, “Localization,” in *SAS/C Library Reference, Volume 2*

“Character Type Macros and Functions” on page 2-1

toupper Translate Lowercase Character to Uppercase

SYNOPSIS

```
#include <ctype.h>

int toupper(int c);
```

DESCRIPTION

`toupper` translates a lowercase character **`c`** to the corresponding uppercase character. The argument must either be a **`char`** value or **`EOF`**. The mapping of lowercase to uppercase characters is locale dependent.

RETURN VALUE

If the argument is a lowercase character, the corresponding uppercase character is returned; otherwise, the argument value is returned.

IMPLEMENTATION

`toupper` is implemented by the compiler as a built-in function, unless you use the name **`toupper`** with **`#undef`**.

EXAMPLE

```
#include <ctype.h>
#include <stdio.h>

#define MAX 40

main()
{
   char *str, *ptr;
   char input[MAX];

   puts("Enter a string of lowercase characters (maximum of 40):");
   str = gets(input);

   ptr = str;

      /* Translate all lowercase characters in a string to      */
      /*  uppercase characters.                                 */
   while (*str) {
      *str = toupper(*str);

         /* Increment outside of macro for maximum portability. */
      str++;
   }
   printf("%s\n", ptr);
}
```

toupper *(continued)* Translate Lowercase Character to Uppercase

RELATED FUNCTIONS

`isupper`, **`memupr`**, **`strupr`**, **`tolower`**

SEE ALSO

Chapter 10, "Localization," in *SAS/C Library Reference, Volume 2*

"Character Type Macros and Functions" on page 2-1

ttyname Get Terminal Name

POSIX UNIX

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

char *ttyname(int fn);
```

DESCRIPTION

ttyname returns the name of the OpenEdition terminal associated with the file descriptor **fn**. The **ttyname** function returns **NULL** if the file descriptor is not open, or if it does not refer to an OpenEdition terminal.

RETURN VALUE

ttyname returns the name of the terminal if it is successful and a NULL pointer if it is not successful.

CAUTION

Subsequent calls to **ttyname** may overwrite the terminal name string.

EXAMPLE

This example determines, for each of the standard POSIX files, whether the file is a terminal and, if so, prints its name:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main() {
   char *name;
   int count = 0;

   name = ttyname(STDIN_FILENO);
   if (name) {
      ++count;
      printf("The standard input is a terminal file named %s\n",
             name);
   }

   name = ttyname(STDOUT_FILENO);
   if (name) {
      ++count;
      printf("The standard output is a terminal file named %s\n",
             name);
   }
```

ttyname Get Terminal Name

(continued)

```
      name = ttyname(STDERR_FILENO);
      if (name) {
         ++count;
         printf("The standard error output is a terminal file named %s\n",
                name);
      }

      if (!count)
         puts("None of the standard files is a terminal file.");
      return 0;
   }
```

RELATED FUNCTIONS

`ctermid`

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

tzset Specify Time Zone

SYNOPSIS

```
#include <time.h>

void tzset(void);
```

DESCRIPTION

tzset accesses the environment variable **TZ**, and uses it to define time-zone information for the other timing functions (**ctime**, **localtime**, **strftime**, and **mktime**). If **TZ** is not defined, the default time zone is implementation defined (see "Timing Functions" on page 2-11).

See "Timing Functions" on page 2-11 for the expected format of the **TZ** environment variable.

In addition to saving time-zone information for the other timing functions, **tzset** stores time-zone names in the external array **tzname**, declared in **<time.h>** as:

```
extern char *tzname[2];
```

tzname[0] is set by **tzset** to the name of the standard time zone, and **tzname[1]** is set to the name of the daylight savings time zone.

Note: The **localtime**, **ctime**, **strftime**, and **mktime** functions call **tzset** themselves during processing. Therefore, you ordinarily do not have to call it yourself.

Note: The prototype for **tzset** in **<time.h>** is not visible unless **_SASC_POSIX_SOURCE** or another POSIX feature test macro is defined before **<time.h>** is included.

RETURN VALUE

There is no return value for **tzset**.

CAUTION

The external variable **tzname** can be accessed only in the main load module of an application. In other load modules, this information can be accessed by calling the function **_tzname()**.

EXAMPLE

This example sets the **TZ** environment variable to Pacific Standard Time and Pacific Daylight Time:

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

#define _SASC_POSIX_SOURCE1

main()
```

tzset Specify Time Zone
(continued)

```
{
   time_t timeval;

   setenv("TZ","PST8PDT");
   tzset();
   time(&timeval);
      /* Print the current Pacific time. */
   printf("The current Pacific time and date are: %s",
          ctime(&timeval));
}
```

RELATED FUNCTIONS

`ctime`, **`localtime`**, **`mktime`**, **`strftime`**

SEE ALSO

"Timing Functions" on page 2-11

ungetc Push Back an Input Character

SYNOPSIS

```
#include <stdio.h>

int ungetc(int c, FILE *f);
```

DESCRIPTION

Use **`ungetc`** to undo the effect of **`getc`**. The **`ungetc`** function backs up the file input pointer so that the next call to an input routine returns **`c`**. The call **`ungetc(EOF, f)`** is valid but has no effect.

For a binary file, a successful call to **`ungetc`** moves the file position back to the previous character, unless the file is positioned at the beginning.

RETURN VALUE

`ungetc` returns **`c`** if its operation was successful or **`EOF`** if **`c`** cannot be pushed back. You may not be able to push back more than a single character, depending on the file contents and attributes.

PORTABILITY

Portable use of **`ungetc`** is limited to one character. Using **`ungetc`** to push back multiple characters without an intervening **`read`** is not portable.

EXAMPLE

This example reads a line from the terminal and separates it into words using **`ungetc`**. Note that this operation could be done more easily using **`scanf`**:

```
#include <stdio.h>
#include <ctype.h>

int wordcnt = 0;

static int skipspace(void);
static int printword(void);

main()
{
   char line[80];

   puts("Enter a short line of text:");
   for(;;) {
      if (skipspace() == 0)
         break;

         /* Skip white space; stop at end of line.            */
      if (printword() == 0)      /* Print the next word.      */
         break;
   }
   if (wordcnt == 0) puts("There were no words in that line.\n");
      exit(0);
}
```

ungetc Push Back an Input Character
(continued)

```
static int skipspace(void) {

   /* Read white space characters from standard input.  Use    */
   /* ungetc() to put back any nonwhite space character        */
   /* found. If a new line is read, stop reading and return 0. */

   int ch;

   for (;;) {
      ch = getchar();
      if (ch == EOF || ch == '\n') return 0;
      if (!isspace(ch)) break;
   }

      /* Put back nonspace for printword to read.              */
   ungetc(ch, stdin);
   return 1;
}

static int printword(void) {
   int ch;
   if (wordcnt == 0)
      puts("Words found in input:");
   ++wordcnt;
   for(;;) {
      ch = getchar();
      if (ch == EOF || ch == '\n') return 0;
      if (!isspace(ch)) putchar(ch);
      else return 1;
   }
}
```

RELATED FUNCTIONS

fgetc, **getc**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

unlink Delete a File

SYNOPSIS

```
#include <fcntl.h>

int unlink(const char *name);
```

The synopsis for the POSIX implementation is

```
#include <sys/types.h>
#include <unistd.h>

int unlink(const char *name)
```

You may use either set of header files in your program.

DESCRIPTION

Under MVS, the **unlink** function deletes the MVS disk file, PDS member, or HFS file specified by the string that is pointed to by **name**.

Under CMS, **unlink** deletes the disk file specified by the CMS string that is pointed to by **name**.

RETURN VALUE

The **unlink** function returns 0 if the file is deleted. If the file cannot be deleted or **name** is invalid, −1 is returned.

CAUTION

MVS
For **ddn** style filenames that do not refer to a PDS member, **unlink** means "to make empty." Other style filenames are deleted and uncataloged.

CMS
If the fileid has a blank filemode, it defaults to **A1**. The **name** function should not contain wild-card values such as ***** or **=**.

POSIX
For an OpenEdition HFS file, the directory entry is deleted. However, the file itself is deleted only when there are no links that refer to it.

IMPLEMENTATION

unlink is an alternate name for **remove**. See the implementation details for **remove**.

EXAMPLE

```
#include <fcntl.h>
#include <stdio.h>

main()
```

unlink Delete a File

(continued)

```
{
   int rc;

   rc = unlink("cms:testfile text a1");

   if (rc == 0)
      puts("The file has been unlinked/deleted.");
   else
      puts("The file could not be deleted.");
}
```

RELATED FUNCTIONS

`link`

SEE ALSO

"File Management Functions" on page 2-16

_unlink Delete an HFS File

DESCRIPTION

`_unlink` is a version of `unlink` designed to operate only on HFS files. `_unlink` runs faster and calls fewer other library routines than `unlink`. Refer to `unlink` for a full description.

`_unlink` is used exactly like the standard `unlink` function. The argument to `_unlink` is interpreted as an HFS filename, even if it appears to begin with a style prefix or a leading `//` or both.

utime Specify Access and Modification Times for a File

POSIX UNIX

SYNOPSIS

```
#include <utime.h>

int utime(char *name, struct utimbuf *time)
```

DESCRIPTION

utime updates the access and modification times for a file. **name** is the filename. **time** is the pointer to a **utimbuf** structure, which contains the new access and modification times. If **time** is **NULL**, the access and modification times for **name** are changed to the current time.

utimbuf contains the following:

time_t actime	is the access time.
time_t modtime	is the modification time.

RETURN VALUE

utime returns 0 if it is successful and −1 if it is not successful.

EXAMPLE

This example sets the modification time for a file to be the same as the time of last access:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <utime.h>
#include <stdlib.h>

main(int argc, char *argv[]) {
   struct stat fdata;
   struct utimbuf tdata;
   int rc;

   if (argc != 2) {
      puts("Incorrect number of arguments");
      exit(EXIT_FAILURE);
   }
   rc = stat(argv[1], &fdata);
   if (rc != 0) {
      perror("stat failure");
      exit(EXIT_FAILURE);
```

utime Specify Access and Modification Times for a File
(continued)

```
      }
      tdata.modtime = tdata.actime = fdata.st_atime;
         /* Set modification time to access time.         */
      rc = utime(argv[1], &tdata);  /* Update file times. */
      if (rc != 0) {
         perror("utime failure");
         exit(EXIT_FAILURE);
      }
      exit(EXIT_SUCCESS);
   }
```

SEE ALSO

Chapter 19, "Introduction to POSIX," in *SAS/C Library Reference, Volume 2*

"File Management Functions" on page 2-16

va_arg Access an Argument from a Varying-Length Argument List

SYNOPSIS

```
#include <stdarg.h>

(arg_type) va_arg(va_list ap, arg_type);
```

DESCRIPTION

va_arg returns the value of the next argument in a varying-length argument list.

The first argument, **ap**, is a work area of type **va_list**, which is used by the expansions of the various **<stdarg.h>** macros. (The **va_list** must be initialized by a previous use of the **va_start** macro, and a corresponding **va_end** must not have been used.)

The second argument, **arg_type**, is the type of the argument that is expected. **arg_type** must be written in such a form that **arg_type *** is the type of a pointer to an element of that type. For example, **char** is a valid **arg_type** because **char *** is the type of a pointer to a character. **int(*)()** is not a valid second argument to **va_arg** because **int(*)()*** is not a valid type. This is not a serious limitation because you can use **typedef** declarations to create usable synonyms of this sort for any type.

If the actual value passed is not of the type specified, the results are unpredictable.

RETURN VALUE

va_arg returns the value of the next argument in the list. The type is always the same as the second argument to **va_arg**.

CAUTION

The results of **va_start** are unpredictable if the argument values are not appropriate.

In certain cases, arguments are converted when they are passed to another type. For instance, **char** and **short** arguments are converted to **int**, **float** to **double**, and array to pointer. When parameters of this sort are expected, **va_arg** must be issued with the type after conversion. For example, **va_arg(ap, float)** may fail to access a **float** argument value correctly, so you should use **va_arg(ap, double)**.

There is no way to test whether a particular argument is the last one in the list. Attempting to access arguments after the last one in the list produces unpredictable results.

va_arg Access an Argument from a Varying-Length Argument List
(continued)

EXAMPLE

This example shows a function named **concat**, which can be used to concatenate any number of strings. A sample call is

```
concat(3, a, b, c);
```

This has the same effect as

```
strcat(a,b);
strcat(a,c);
```

(The first argument is the total number of strings.)

```
#include <stdarg.h>
#include <string.h>

void concat(int count, ...)
{
  va_list ap;
  char *target, *source;
  int i;

  if (count <= 1) return;

  va_start(ap, count);

  target = va_arg(ap, char *); /* Get target string.       */
  target += strlen(target);    /* Point to string end.     */

  while(--count > 0) {

        /* Get next source string.                         */
     source = va_arg(ap, char *);

                               /* Copy chars to target.    */
     while(*source) *target++ = *source++;
     }
  *target = '\0';              /* Add final null.          */
  va_end(ap);                  /* End arg list processing. */
  return;
}
```

RELATED FUNCTIONS

va_end, **va_start**

SEE ALSO

"Varying-Length Argument List Functions" on page 2-9

va_end End Varying Text-Length Argument List Processing

SYNOPSIS

```
#include <stdarg.h>

void va_end(va_list ap);
```

DESCRIPTION

`va_end` completes processing of a varying-length argument list. The argument **`ap`** is a work area of type **`va_list`**, which is used by the expansions of the various **`<stdarg.h>`** macros.

After **`va_end`** is called, **`va_start`** must be called again before you can use **`va_arg`**.

RETURN VALUE

`va_end` has no return value.

In this implementation, using **`va_end`** in varying-length argument list processing is not required. However, in other implementations, failure to issue **`va_end`** may cause program failures on return from the function that issued **`va_start`**.

EXAMPLE

See the example for **`va_arg`**.

RELATED FUNCTIONS

`va_arg`, **`va_start`**

SEE ALSO

"Varying-Length Argument List Functions" on page 2-9

va_start Begin Varying-Length Argument List Processing

SYNOPSIS

ISO ANSI

```
#include <stdarg.h>

void va_start(va_list ap, arg_name);
```

DESCRIPTION

`va_start` initializes processing of a varying-length argument list. The first argument, **`ap`**, is a work area of type **`va_list`**, which is used by the expansions of the various **`<stdarg.h>`** macros. The second argument, **`arg_name`**, is the name of the parameter to the calling function after which the varying part of the parameter list begins.

This function is one of three macros used to advance through a list of arguments whose number and type are unknown when the function is compiled. The other two macros are

`va_arg`	accesses an argument from a varying-length argument list.
`va_end`	ends varying-length argument list processing.

These macros and the type **`va_list`** are defined in the header file **`<stdarg.h>`**.

The type **`va_list`** defines a buffer that is used as a work area during argument list processing. A routine that accepts a varying number of arguments must declare an **`auto`** variable of this type.

In general, a function that uses the **`<stdarg.h>`** facilities has this form:

```
#include <stdarg.h>

   /* The arguments in the list are the ones that must */
   /*  always be present.                              */
func(type  arg1, type  arg2)
{
   va_list ap;      /* Declare stdarg work area.       */

      /* Note that first varying-length argument       */
      /*  follows arg2 in the list.                    */
   va_start(ap, arg2);

   while (more_args_expected) {

         /* Get next argument value.                   */
      this_arg = va_arg(ap, type);
      process(this_arg);
   }
   va_end(ap);      /* finished argument processing    */
}
```

RETURN VALUE

`va_start` has no return value.

va_start *(continued)*

Begin Varying-Length Argument List Processing

CAUTION

The results of `va_start` are unpredictable if the argument values are not appropriate.

EXAMPLE

See the example for `va_arg`.

RELATED FUNCTIONS

`va_arg`, `va_end`

SEE ALSO

"Varying-Length Argument List Functions" on page 2-9

vfprintf Write Formatted Output to a File

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vfprintf(FILE *f, const char *format, va_list arg);
```

DESCRIPTION

vfprintf is equivalent to **fprintf** with **arg** replacing the variable-argument list. **arg** has been initialized by the **va_start** macro and possibly **va_arg** calls. **vfprintf** does not invoke the **va_end** macro. See **va_arg**, **va_end**, and **va_start** for details on varying-length argument-list functions.

RETURN VALUE

vfprintf returns the number of characters transmitted to the output stream or a negative value if an output error occurs.

EXAMPLE

This example sends an error message prefix with **fprintf** and sends the remaining text with **vfprintf**:

```
#include <stdarg.h>
#include <stdio.h>

void error(char *fname, char *format, ...)
{
   va_list args;
   va_start(args, format);
   fprintf(stderr, "ERROR in %s: ", fname);
   vfprintf(stderr, format, args);
   va_end(args);
}
```

RELATED FUNCTIONS

fprintf, **va_start**, **vprintf**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

vprintf Write Formatted Output to the Standard Output Stream

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vprintf(const char *format, va_list arg);
```

DESCRIPTION

`vprintf` is equivalent to **`printf`** with **`arg`** replacing the variable-argument list. **`arg`** has been initialized by the **`va_start`** macro and possibly **`va_arg`** calls. **`vfprintf`** does not invoke the **`va_end`** macro. See **`va_arg`**, **`va_end`**, and **`va_start`** for details on varying-length argument-list functions.

RETURN VALUE

`vprintf` returns the number of characters transmitted to **`stdout`** or a negative value if an output error occurs.

EXAMPLE

This example sends an error message prefix to **`stdout`** with **`printf`** and sends the remaining text to **`stdout`** with **`vprintf`**:

```
#include <stdarg.h>
#include <stdio.h>

void error(char *fname, char *format, ...)
{
   va_list args;
   va_start(args, format);
   printf("ERROR in %s: ", fname);
   vprintf(format, args);
   va_end(args);
}
```

RELATED FUNCTIONS

`printf`, **`va_start`**, **`vfprintf`**

SEE ALSO

Chapter 3, "I/O Functions" on page 3-1

"I/O Functions" on page 2-13

vsnprintf Write a Limited Portion of Formatted Output to a String

SYNOPSIS

```
#include <stdarg.h>
#include <lcio.h>

int vsnprintf(char *dest, size_t maxlen, const char *format,
              va_list arg);
```

DESCRIPTION

The **vsnprintf** function is equivalent to the **snprintf** function, except that **arg** replaces the variable-argument list. **arg** must have been initialized by the **va_start** macro and possibly **va_arg** calls. The **vsnprintf** function does not invoke the **va_end** macro. See **va_arg**, **va_end**, and **va_start** for details on varying-length argument-list functions. The **vsnprintf** function writes formatted output to the area addressed by **dest** under control of the string addressed by **format** until either all format conversion specifications have been satisfied or until **maxlen** characters have been written. If the **maxlen** limit is reached

- a terminating-null character is not added
- the number of characters placed in the output area will be the value of **maxlen**
- the remainder of the format string is ignored
- The **vsnprintf** function returns a negative value whose magnitude is equal to the value of **maxlen**.

In all other respects, **vsnprintf** behaves identically to **vsprintf**. The string pointed to by format is in the same form as that used by **fprintf**. Refer to the **fprintf** function description for detailed information concerning the format conversions.

RETURN VALUE

The **vsnprintf** function returns an integer value that equals in magnitude the number of characters written to the area addressed by **dest**. If the value returned is negative, then either the **maxlen** character limit was reached or some other error, such as an invalid format specification, has occurred. The one exception to this is if an error occurs before any characters are stored, **vsnprintf** returns INT_MIN (-2**31).

CAUTION

If the **maxlen** value is 0, no characters are written, and **vsnprintf** returns 0. If the value is greater than **INT_MAX**, then **vsnprintf** behaves identically to **vsprintf** in that no limit checking is done on the number of characters written to the output area. No warnings concerning length errors are produced by **vsnprintf**, and the only indication that the output may have been truncated or is incomplete is a negative return value.

vsnprintf Write a Limited Portion of Formatted Output to a String
(continued)

IMPLEMENTATION

When invoked with a limit greater than 512 characters, the **vsnprintf** function calls **malloc** to obtain a temporary spill buffer equal in size to the limit specified. If insufficient storage is available, **vsnprintf** attempts to process the format specifications with an internal 512-byte spill buffer. In this case, individual conversion specifiers that produce more than 512 characters may fail, and **vsnprintf** processing can terminate prematurely.

EXAMPLE

```
#include <stdlib.h>
#include <lcio.h>
#include <stdarg.h>

static void format_msg(char *, size_t, char *, ...);

#define NOTE 0
#define WARNING 1
#define MESSAGE_LEN 80

static const char *msgs[] = {
  "Msgno%04d  This is message number zero",
  "Msgno%04d  This message requires a %s"
};

main()
{
   char msgbuf(|84|);

   format_msg(msgbuf, MESSAGE_LEN, msgs[NOTE], NOTE);
   printf("The formatted message is: \"%s\" \n", msgbuf);
   format_msg(msgbuf, MESSAGE_LEN, msg[WARNING], WARNING,
              "a replacement string");
   printf("The formatted message is: \"%s\" \n", msgbuf);
   return;
}

static void format_msg(char *buf, size_t limit, char *format,...)
{
   va_list args;
   va_start(args, format),
   vsnprintf(buf, limit, format, args);
   va_end(args);
}
```

vsnprintf Write a Limited Portion of Formatted Output to a String
(continued)

RELATED FUNCTIONS

`snprintf`, `va_start`, `vsprintf`

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

vsprintf Write Formatted Output to a String

ISO ANSI

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vsprintf(char *s, const char *format, va_list arg);
```

DESCRIPTION

vsprintf is equivalent to **sprintf** with **arg** replacing the variable-argument list. **arg** has been initialized by the **va_start** macro and possibly **va_arg** calls. **vsprintf** does not invoke the **va_end** macro. See **va_arg**, **va_end**, and **va_start** for details on varying-length argument-list functions.

RETURN VALUE

vsprintf returns the number of characters written in the array, not counting the terminating-null character.

EXAMPLE

```
#include <stdio.h>
#include <stdarg.h>
#include <string.h>

static void format_msg(char *, int, int, ...);

#define NOTE 0
#define WARNING 1

main()
{
   char msgbuf[100];
   format_msg(msgbuf, NOTE, 0);
   printf("The formatted message is: \"%s\" \n", msgbuf);
   format_msg(msgbuf, WARNING,1, "a replacement string");
   printf("The formatted message is: \"%s\" \n", msgbuf);
   return;
}

static const char *msgs[]  = {
   "This is message number zero",
   "This message requires %s"
};

static const char *levels[]  = {
   "NOTE: ",
   "WARNING: "
};

static void format_msg(char *buf, int msgno, int level, ...)
```

vsprintf Write Formatted Output to a String
(continued)

```
{
   va_list args;
   va_start(args, level);

      /* Copy in the message prefix. Format the     */
      /*  remainder of the message with vsprintf(). */
   strcpy(buf, levels[level]);
   vsprintf(buf + strlen(levels[level]),msgs[msgno], args);
   va_end(args);
}
```

RELATED FUNCTIONS

sprintf, **va_start**, **vformat**, **vsnprintf**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

write Write Data to a File or Socket

SYNOPSIS

```
#include <fcntl.h>

int write(int fn, const void *buf, size_t size);
```

The synopsis for the POSIX implementation is

```
#include <unistd.h>

ssize_t write(init fn, const void *buf, size_t size)
```

DESCRIPTION

`write` writes data to the file or socket with file descriptor **`fn`** from the buffer addressed by **`buf`**. The number of bytes written is **`size`**. If **`fn`** is associated with a socket, it must either be connected or have been assigned an associated remote address by the **`connect`** function.

If **`size`** is 0, **`write`** returns a value of zero and does not attempt any other operation. Even though the **`size`** argument is defined as a **`size_t`**, the maximum value permitted is **`INT_MAX`**, the largest signed integer.

RETURN VALUE

`write` returns the number of bytes written or −1 if an error occurs.

EXAMPLE

```
#include <fcntl.h>

double matrix[100][100];
int tempfile;
.
.
.
write(tempfile, matrix, sizeof(matrix));
```

RELATED FUNCTIONS

`fwrite`, **`writev`**

SEE ALSO

Chapter 3, “I/O Functions” on page 3-1

“I/O Functions” on page 2-13

_write Write Data to an HFS File

DESCRIPTION

`_write` is a version of **`write`** designed to operate only on HFS files. **`_write`** runs faster and calls fewer other library routines than **`write`**. Refer to **`write`** for a full description.

`_write` is used exactly like the standard **`write`** function. The first argument to **`_write`** must be the file descriptor for an open HFS file.

xltable Build Character Translation Table

SYNOPSIS

```
#include <lcstring.h>

char *xltable(char table[256], char *source, char *target);
```

DESCRIPTION

`xltable` builds a translation table that you can use later as an argument to the **`memxlt`** or **`strxlt`** function.

The argument **`table`** is a 256-character array, in which the translation table is to be built. The second argument (**`source`**) is a string of characters that the table is to translate, and the third argument (**`target`**) is a string containing the characters to which the source characters are to be translated, in the same order. The **`source`** and **`target`** strings should contain the same number of characters; if they do not, the extra characters of the longer string are ignored.

You can also specify a table address of 0. In this case, **`xltable`** builds the table in a static area and returns the address of this area. This area may be overlaid by the next call to **`xltable`**, **`strscntb`**, or **`memscntb`**.

The table built by **`xltable`** translates any character not present in the **`source`** string to itself, so these characters are not changed when using the table.

RETURN VALUE

`xltable` returns the address of the table (the same value as that of the first argument).

EXAMPLE

See **`strxlt`** and **`memxlt`** for examples.

RELATED FUNCTIONS

`memxlt`, **`strxlt`**

SEE ALSO

"String Utility Functions" on page 2-3

y0 Bessel Function of the Second Kind, Order 0

UNIX

SYNOPSIS

```
#include <lcmath.h>

double y0(double x);
```

DESCRIPTION

y0 computes the order 0 Bessel function of the second kind of the value **x**.

RETURN VALUE

y0 returns the order 0 Bessel function of the second kind of the argument **x**, provided that this value is computable.

DIAGNOSTICS

If the value of **x** is 0.0, a diagnostic message is written to the standard-error file (**stderr**) and the function returns **-HUGE_VAL**, the largest negative floating-point number that can be represented.

If the magnitude of **x** is too large ($|x| >= 6.7465e9$), **y0** returns 0.0. In this case, the message “total loss of significance” is written to **stderr**.

If an error occurs in **y0**, the **_matherr** routine is called. You can supply your own version of **_matherr** to suppress the diagnostic message or modify the value returned.

EXAMPLE

This example computes the Bessel function of the second kind, of order 0 at x = 5:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;
   y = y0(5.);
   printf("y0(5.) = %lf\n", y);
}
```

RELATED FUNCTIONS

j0, **j1**, **jn**, **_matherr**, **y1**, **yn**

SEE ALSO

“Mathematical Functions” on page 2-6

y1 Bessel Function of the Second Kind, Order 1

UNIX

SYNOPSIS

```
#include <lcmath.h>

double y1(double x);
```

DESCRIPTION

y1 computes the order 1 Bessel function of the second kind of the value **x**.

RETURN VALUE

y1 returns the order 1 Bessel function of the second kind of the argument **x**, provided that this value is computable.

DIAGNOSTICS

If the value of **x** is 0.0, a diagnostic message is written to the standard error file (**stderr**) and the function returns **-HUGE_VAL**, the largest negative floating-point number that can be represented.

If the magnitude of **x** is too large (|x| >= 6.7465e9), **y1** returns 0.0. In this case, the message “total loss of significance” is written to **stderr**.

If the magnitude of **x** is too close to 0 (|x| < approximately 8.032e-77), an overflow error occurs during computation of **y1**. In this case, a diagnostic message is written to **stderr**, and **y1** returns **-HUGE_VAL**.

If an error occurs in **y1**, the **_matherr** routine is called. You can supply your own version of **_matherr** to suppress the diagnostic message or modify the value returned.

EXAMPLE

This example computes the Bessel function of the second kind, of order 1 at x = 5:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;
   y = y1(5.);
   printf("y1(5.) = %lf\n", y);
}
```

RELATED FUNCTIONS

j0, **j1**, **jn**, **_matherr**, **y0**, **yn**

SEE ALSO

“Mathematical Functions” on page 2-6

yn Bessel Function of the Second Kind, Order n

SYNOPSIS

```
#include <lcmath.h>

double yn(int n, double x);
```

DESCRIPTION

yn computes the order **n** Bessel function of the second kind of the value **x**.

The CPU time required to compute the Bessel function increases with increasing values for **n**. For very large values of **n**, the time can be quite large.

RETURN VALUE

yn returns the order **n** Bessel function of the second kind of the argument **x**, provided that this value is computable.

DIAGNOSTICS

If the value of **x** is 0.0, a diagnostic message is written to the standard error file (**stderr**) and the function returns **-HUGE_VAL**, the largest negative floating-point number that can be represented.

If the magnitude of **x** is too large (|x| >= 6.7465e9), **yn** returns 0.0. In this case, the message “total loss of significance” is written to **stderr**.

If the magnitude of **x** is too close to 0, an overflow error occurs during computation of **yn**. The limiting value for **x** depends on the value for **n**. If **n** is 1, the limiting value is approximately 8.032e-77. The limiting value increases with increasing values of **n**. When **x** is too small, a diagnostic message is written to **stderr**, and **yn** returns **-HUGE_VAL**.

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

EXAMPLE

This example computes the Bessel function of the second kind, of order 7 at x = 5:

```
#include <stdio.h>
#include <lcmath.h>

main()
{
   double y;
   y = yn(7, 5.);
   printf("yn(7, 5.) = %lf\n", y);
}
```

yn Bessel Function of the Second Kind, Order n
(continued)

RELATED FUNCTIONS

`j0`, `j1`, `jn`, `_matherr`, `y0`, `y1`

SEE ALSO

"Mathematical Functions" on page 2-6

Function Index