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

Volume 2

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 2, Release 6.00*, Cary, NC: SAS Institute Inc., 1995. 623 pp.

SAS/C® Library Reference, Third Edition, Volume 2, Release 6.00

ISBN 1-55544-667-1

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

1st printing, October 1995

Doc P17, 092595

Contents

Part 1 Special Features of the SAS/C Library

1-1 Chapter 1 **Dynamic-Loading Functions**
1-1 Introduction
1-2 Function Descriptions

2-1 Chapter 2 **CMS Low-Level I/O Functions**
2-1 Introduction
2-2 CMS Low-Level I/O Functions
2-5 XEDIT Low-Level I/O Functions
2-8 Related Utility Functions

3-1 Chapter 3 **MVS Low-Level I/O Functions**
3-1 Introduction
3-2 DCBs and DCB Exit Routines
3-3 Direct BSAM Interface Functions
3-9 The Record-Oriented BSAM Interface
3-11 BSAM Record-Oriented Interface Functions
3-16 The osdynalloc Function

4-1 Chapter 4 **MVS Multitasking and Other Low-Level System Interfaces**
4-1 Introduction
4-1 Multitasking SAS/C Applications
4-2 Compatibility Changes
4-3 Function Descriptions

5-1 Chapter 5 **Inter-User Communications Vehicle (IUCV) Functions**
5-1 Introduction
5-1 IUCV Communications Overview
5-3 IUCV, Signal Handling, and Message Processing
5-5 IUCV Parameter Lists and External Interrupt Data Formats
5-7 Message Queues
5-8 Return Codes
5-8 Function Descriptions
5-15 An Example of IUCV Communication
5-21 Guidelines and Cautions

6-1 Chapter 6 **Advanced Program-to-Program Communication/Virtual Machine (APPC/VM) Functions**
6-1 Introduction
6-2 APPC/VM Parameter Lists and External Interrupt Data Formats
6-4 Function Descriptions
6-8 APPC/VM Communication Example Programs
6-16 Guidelines and Cautions

7-1 Chapter 7 **The Subcommand Interface to EXECs and CLISTs**
7-1 Introduction
7-1 Subcommand Processing in C Programs
7-3 Overview of the SUBCOM Environment
7-5 Function Descriptions
7-27 Examples of SUBCOM Processing
7-32 Guidelines for Subcommand Processing

8-1 Chapter 8 **The CMS REXX SAS/C Interface**
8-1 Introduction
8-1 REXX Concepts and Background
8-3 How the Library Interfaces with REXX
8-6 Function Descriptions
8-17 An Example of a REXX Function Package
8-19 Additional Guidelines and Related Topics

9-1 Chapter 9 **Coprocessing Functions**
9-1 Introduction
9-2 cocall and coreturn
9-3 Coprocess States
9-4 Passing Data between Coprocesses
9-5 Coprocess Data Types and Constants
9-6 Coprocess Identifiers
9-6 Restrictions
9-6 A Coprocessing Example
9-9 Advanced Topics: Program Termination
9-10 Advanced Topics: Signal Handling
9-11 Function Descriptions

10-1 Chapter 10 **Localization**
10-1 Introduction
10-1 Locales and Categories
10-4 The “S370” Locale
10-4 The “POSIX” Locale
10-4 Library-Supplied Locales
10-5 Function Descriptions
10-17 User-Added Locales

11-1 Chapter 11 **Multibyte Character Functions**
11-1 Introduction
11-1 SAS/C Implementation of Multibyte Character Sequences
11-4 Locales and Multibyte Character Sequences
11-4 Function Descriptions

12-1 Chapter 12 **User-Added Signals**
12-1 Introduction
12-2 Notes on Handlers for User-Defined Signals
12-2 Restrictions on Synchronous and Asynchronous Signals
12-2 Summary of Routines for Adding Signals
12-8 SAS/C Library Routines for Adding New Signals

13-1 Chapter 13 **Getting Environmental Information**
13-1 Introduction
13-1 The L$UENVR Routine
13-3 Function Descriptions

Part 2 SAS/C Socket Library for TCP/IP

14-1 Chapter 14 **The TCP/IP Protocol Suite**
14-1 Overview of TCP/IP
14-2 Internet Protocol (IP)
14-2 User Datagram Protocol (UDP)
14-3 Transmission Control Protocol (TCP)
14-3 Domain Name System (DNS)
14-4 MVS and CMS TCP/IP Implementation

15-1 Chapter 15 **The BSD UNIX Socket Library**
15-1 Introduction
15-2 Overview of the BSD UNIX Socket Library
15-3 Header Files
15-12 Socket Functions

16-1 Chapter 16 **Porting UNIX Socket Applications to the SAS/C Environment**
16-1 Introduction
16-1 Integrated and Non-Integrated Sockets
16-2 Socket Library Restrictions
16-3 Function Names
16-4 Header Filenames
16-4 errno
16-4 BSD Library Dependencies
16-5 INETD and BSD Kernel Routine Dependencies
16-6 Character Sets
16-6 The Resolver

17-1 Chapter 17 **Network Administration**
17-1 Introduction
17-1 Configuration Data Sets
17-1 Search Logic
17-3 Specifying TCPIP_PREFIX for MVS
17-4 Specifying TCPIP_MACH
17-4 gethostbyname and Resolver Configuration
17-5 Configuring for the CMS Environment
17-5 Configuring for the MVS Environment

18-1 Chapter 18 **Socket Function Reference**
18-1 Introduction

Part 3 SAS/C POSIX Support

19-1 Chapter 19 **Introduction to POSIX**
19-1 POSIX and OpenEdition Concepts
19-3 The errno Variable
19-6 SAS/C OpenEdition Interfaces

20-1 Chapter 20 **POSIX Function Reference**
20-1 Introduction

Function Index

Index

Using This Book

SAS/C Library Reference, Third Edition, Volume 2, 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 C Conforming

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

POSIX.1 Conforming

These functions conform 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: 919-677-8166
E-mail: sasbook1@vm.sas.com

Online Documentation

This book is also available in html format for on-line 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 2, 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 #A55049) 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

Special Features of the SAS/C® Library

Chapters

1 Dynamic-Loading Functions
2 CMS Low-Level I/O Functions
3 MVS Low-Level I/O Functions
4 MVS Multitasking and Other Low-Level System Interfaces
5 Inter-User Communications Vehicle (IUCV) Functions
6 Advanced Program-to-Program Communication/Virtual Machine (APPC/VM) Functions
7 The Subcommand Interface to EXECs and CLISTs
8 The CMS REXX SAS/C® Interface
9 Coprocessing Functions
10 Localization
11 Multibyte Character Functions
12 User-Added Signals
13 Getting Environmental Information

1 Dynamic-Loading Functions

1-1 Introduction
1-1 Cautions
1-1 Dynamic Load Modules
1-2 Function Descriptions
1-3 CMS Argument Values

Introduction

Although most C programs are linked together as a single load module, for large applications it is often convenient to divide a program into several load modules that can be loaded into memory and unloaded independently. The SAS/C Compiler and Library support multiple load module programs, but you must develop such programs carefully.

As with any C program, program execution always begins with a function named **main**. The load module containing the **main** function must remain in memory at all times. Each subordinate (dynamically loaded) load module must contain a function named **_dynamn** (which is a contraction for *dynamic main*). From the perspective of the compiler, the **_dynamn** function serves the same role in the subordinate load module as **main** serves in the module containing the **main** program.

Cautions

Note that load modules linked with the SAS/C Library are of two distinct types: main modules and dynamically loadable modules. A *main module* is one which contains a **main** function, or which can cause the C environment to be created in conjunction with using the **indep** compiler option. A *dynamically loadable module* is one which contains a **_dynamn** function. It is not possible to create a load module which is of both types. Any attempt to do so will fail at link time or at execution time. Two consequences of this requirement are:

- You cannot link a **main** function and a **_dynamn** function into the same load module. In particular, this means that you cannot have a source file which includes both these functions.
- You cannot create a C framework by calling a function that was compiled with the **indep** option into a load module which includes a **_dynamn** function.

This restriction is imposed so that dynamically loadable modules are not forced to include the large amount of code needed to create a new C framework.

Dynamic Load Modules

Load modules subordinate to the **main** program module can be loaded by use of the **loadm** and **unloadm** functions. The second argument to **loadm** is a pointer to a function pointer in which the address of the loaded **_dynamn** routine will be stored.

In addition to **loadm** and **unloadm**, the library provides functions to load and unload modules containing only data (**loadd** and **unloadd**) and a function that converts a load module entry point address to a function pointer (**buildm**). Two other functions, **addsrch** and **delsrch**, are provided, primarily for CMS, to define the load module *search order*. The search order consists of the possible locations of dynamically loaded modules and the order in which they are processed. Before any of these routines can be used, the source program must include the header file **<dynam.h>** using the **#include** statement.

Transfers of control between load modules are possible only by using function pointers. However, through the use of appropriate pointers, a routine in one load module can call a routine in any other load module. The inability of one load

module to call another directly is a special case of a more general restriction; namely, load modules cannot share external variables. More precisely, two external variables (or functions) of the same name in different load modules are independent of each other. (There are a few special cases such as the variable **errno**, which contains the number of the most recent run-time problem.) See Appendix 4, "Sharing extern Variables among Load Modules," in the *SAS/C Compiler and Library User's Guide* for additional information. An external variable or function can be accessed directly only by functions that have been linked in that module. All functions in other modules can gain access to them only by use of pointers.

The functions for dynamic loading, especially **addsrch** and **delsrch**, are highly operating-system-dependent. The definition of each dynamic-loading function highlights aspects of the function that depend in some way on the operating system. **addsrch** and **delsrch**, for example, are of interest to users working under MVS only when a program needs to be portable to CMS. For CMS, these functions are quite useful but are not needed typically in an MVS environment.

Function Descriptions

Descriptions of each dynamic-loading function follow. Each description includes a synopsis and description, discussions of return values and portability issues, and an example. Also, errors, cautions, diagnostics, implementation details, and usage notes are included if appropriate.

addsrch Indicate a "Location" from which Modules May Be Loaded

SYNOPSIS

```
#include <dynam.h>

SEARCH_P addsrch(int type, const char *loc,
                 const char *prefix);
```

DESCRIPTION

addsrch adds a "location" to the list of "locations" from which modules can be loaded. This list controls the search order for load modules loaded via a call to **loadm**. The search order can be described additionally by the third argument, **prefix**.

The first argument **type** must be a module type defined in **<dynam.h>**. The module type defines what type of module is loaded and varies from operating system to operating system. The character string specified by the second argument **loc** names the location. The format of this string depends on the module type. The third argument **prefix** is a character string of no more than eight characters.

addsrch is of interest primarily to CMS users and to MVS users writing programs portable to CMS. The remainder of this discussion, therefore, focuses on the use of **addsrch** under CMS.

CMS Argument Values

Under CMS, the defined module types for the first argument are the following:

- CMS_NUCX specifies that the module is a nucleus extension. The module has been loaded (for example, by the CMS command NUCXLOAD) before **loadm** is called.
- CMS_LDLB specifies that the module is a member of a CMS LOADLIB file. The LOADLIB file must, of course, be on an accessed disk when **loadm** is called.
- CMS_DCSS specifies that the module resides in a named segment that has been created using the GENCSEG utility, as documented in Appendix 3, "The CMS GENCSEG Utility," in the *SAS/C Compiler and Library User's Guide*.

The module type also controls the format of the second argument **loc**, which names the location to be searched by **loadm**. If the module type is

- CMS_NUCX, the location parameter must be ''''.
- CMS_LDLB, the location parameter is the filename and filemode of the LOADLIB file in the form *filename fm*, for example, DYNAMC A1 specifies the file DYNAMC LOADLIB A1. The filemode may be *.
- CMS_DCSS, the location parameter is a one- to eight-character string that names the segment. An asterisk as the first character in the name is used to specify that the segment name is for a non-shared segment.

All location strings may have leading and trailing blanks. The characters are uppercased. **addsrch** does not verify the existence of the location.

The third argument is a character string of no more than eight characters. It may be ''''. If it is not null, then it specifies that the location indicated is searched only if the load module name (as specified by the first argument to **loadm**) begins with the same character or characters specified in the third argument.

addsrch Indicate a "Location" from which Modules May Be Loaded *(continued)*

At C program initialization, a default location, defined by the following call, is in effect:

```
sp = addsrch(CMS_LDLB, "DYNAMC *","");
```

RETURN VALUE

addsrch returns a value that can be passed to **delsrch** to delete the input source. Under CMS, this specifically means a value of the defined type **SEARCH_P**, which can be passed to **delsrch** to remove the location from the search order. If an error occurs, a value of **0** is returned.

CAUTION

The above arguments to **addsrch** are defined only under CMS. The use of **addsrch** under MVS with a CMS module type has no effect.

USAGE NOTES

addsrch does not verify that a location exists (DYNAMC LOADLIB, for example) or that load modules may be loaded from that location. The **loadm** function searches in the location only if the load module cannot be loaded from a location higher in the search order. **addsrch** fails only if its parameters are ill-formed.

EXAMPLE

```
#include <dynam.h>

SEARCH_P mylib;
.
.
.
   /* Search for modules in a CMS LOADLIB. */
mylib = addsrch(CMS_LDLB, "PRIVATE *", "");
```

buildm Create a Function Pointer from an Address

SYNOPSIS

```
#include <dynam.h>

void buildm(const char *name, __remote /* type */ (**fpp)(),
            const char *ep);
```

DESCRIPTION

buildm converts the entry point address in **ep** to a **__remote** function pointer. The created function pointer can then be used to transfer control to a function or module located at this address. **buildm** is normally used to generate a function pointer for a C load module that has been loaded without the assistance of the SAS/C Library (for instance, by issuing the MVS LOAD SVC), but it can also be used with a non-C load module or with code generated by the program. **buildm** also assigns a load module name to the entry point address, and use of this name in subsequent calls to **loadm** or **unloadm** is recognized as referring to the address in **ep**. Note that a load module processed with **buildm** should always include a **_dynamn** function.

buildm stores the function pointer in the area addressed by **fpp**. Note that **fpp** may reference a function returning any valid type of data. If the function pointer cannot be created, a **NULL** value is stored.

name points to a name to be assigned to the built load module. If **name** is **''**, then a unique name is assigned by **buildm**. If the name is prefixed with an asterisk, then **buildm** does not check to see if the name is the name of a previously loaded module (see "ERRORS", below).

RETURN VALUE

buildm stores the function pointer in the area addressed by **fpp**. If an error occurs, **buildm** stores **NULL** in this area.

ERRORS

If the string addressed by **name** does not start with an asterisk and is the same as a previously built or dynamically loaded module, the request is rejected unless the value of **ep** is the same as the entry point of the existing load module. If the entry points are the same, a pointer to the previously loaded or built module is stored in the area addressed by **func**.

CAUTIONS

The **name** argument must point to a null-terminated string no more than eight characters long, not counting a leading asterisk. Leading and trailing blanks are not allowed.

The **fpp** argument must be a pointer to an object declared as "pointer to function returning (some C data type)".

EXAMPLE

This example illustrates a method of using inline machine code (see Chapter 13, "In-Line Machine Code Interface," in *SAS/C Compiler and Library User's Guide*) to load a load module under MVS or a TEXT file or TXTLIB member

buildm Create a Function Pointer from an Address
(continued)

under CMS using SVC 8. The machine code instruction sequence is coded as a macro to enhance readability.

The example assumes that SIMPLE is a C **_dynamn** function returning **void**.

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

#define LOAD(n,ep) (_ldregs(R0+R1,n,0),_ossvc(8),
        _stregs(R0,ep))

main()
{

   void (*fp)();
   char *ep;

      /* The name "SIMPLE" must be uppercased, left-adjusted, */
      /* and padded to eight characters with blanks when      */
      /* used by SVC 8.                                       */
   LOAD("SIMPLE  ",&ep);

      /* The name passed to buildm does not have to match     */
      /* the name of the loaded module, but it helps.         */
   buildm("simple",&fp,ep);
   if (fp)         /* If no errors, call SIMPLE               */
      (*fp)();
   else
      puts("simple didn't load.");

}
```

delsrch Delete a "Location" in the Load Module Search Order List

SYNOPSIS

```
#include <dynam.h>
void delsrch(SEARCH_P sp);
```

DESCRIPTION

delsrch removes the "location" **sp** pointed to by the argument from the load module search order list. **sp** is a value returned previously by **addsrch**.

PORTABILITY

delsrch is not portable. **delsrch** is used primarily in a CMS environment as a counterpart to **addsrch** or by MVS programs that can port to CMS.

EXAMPLE

The following example illustrates the use of **delsrch** under CMS:

```
#include <dynam.h>

SEARCH_P source;
char *new_source;
.
.
.
   /* Delete old search location.  */
if (source) delsrch(source);
   /* Add new search location.     */
source = addsrch(CMS_LDLB, new_source, "");
```

loadd Dynamically Load a Load Module Containing Data

SYNOPSIS

```
#include <dynam.h>

void loadd(const char *name, char **dp, MODULE *mp);
```

DESCRIPTION

loadd is similar to **loadm** (load executable module) except that it is intended for data modules. **loadd** loads the module named by the argument **name** and stores the address of the load module's entry point in the location pointed to by the second argument **dp**.

If the module has been loaded already, the pointer returned in the second argument points to the previously loaded copy. If the module name in the first argument string is prefixed with an asterisk, a private copy of the module is loaded.

The third argument addresses a location where a value is stored that can be used later to remove the module from memory via **unloadd**. **loadd** should be used only to load modules that contain data (for example, translation tables) rather than executable code.

RETURN VALUE

loadd indirectly returns a value that is stored in the location addressed by the third argument **mp**. This value can be used later to remove the module from memory via **unloadd**. If the module to be loaded cannot be found, **0** is returned.

ERRORS

Various user ABENDs, notably 1217 and 1218, may occur if overlays of library storage are detected while dynamic loading is in progress.

CAUTION

The first argument string may be no more than eight characters long, not counting a leading asterisk. Also note that a module coded with **loadd** must contain at least 16 bytes of data following the entry point, or library validation of the module may fail.

PORTABILITY

loadd is not portable. As with other dynamic-loading functions, be aware of system-specific requirements for the location of modules to be loaded.

IMPLEMENTATION

The implementation of **loadd** necessarily varies from operating system to operating system. Under MVS, modules to be loaded must reside in STEPLIB or the system link list. Under CMS, modules to be loaded may reside in DYNAMC LOADLIB or in other locations defined by use of the **addsrch** routine.

Under CICS, modules to be loaded must reside in a library in the DFHRPL concatenation, and must be defined to CICS.

loadd Dynamically Load a Load Module Containing Data
(continued)

EXAMPLE

The following example illustrates a general case of using **loadd** :

```
#include <dynam.h>
#include <lcstring.h>

char *table;
char *str;
MODULE tabmod;
   /* Load a translate table named LC3270AE.                    */
loadd("LC3270AE",&table,&tabmod);
str = strxlt(str, table);
unloadd(tabmod);                      /* Unload module after use.*/
```

loadm Dynamically Load a Load Module

SYNOPSIS

```
#include <dynam.h>;

void loadm(const char *name, __remote /* type */ (**fpp)());
```

DESCRIPTION

loadm loads an executable module named by the argument string **name** and stores a C function pointer in the location pointed to by the argument **fpp**. If the module has been loaded already, the pointer stored in **fpp** points to the previously loaded copy. If the module name in the first argument string is prefixed with an asterisk, a private copy of the module is loaded. Note that **fpp** may reference a function returning any valid type of data.

RETURN VALUE

loadm provides an indirect return value in the form of a function pointer that addresses the entry point of the loaded module. If the module is in C, calling the returned function always transfers control to the **_dynamn** function of the module.

If the module to be loaded cannot be found, a **NULL** is stored in the location addressed by **fpp**.

ERRORS

Various user ABENDs, notably 1217 and 1218, may occur if overlays of library storage are detected while dynamic loading is in progress.

CAUTIONS

The first argument string may be no more than eight characters long, not counting a leading asterisk.

The second argument must be a pointer to an object declared as "pointer to function returning (some C data type)."

Note that a module to be loaded by **loadm** cannot have the entry point defined in the last 16 bytes of the load module. The library inspects this portion of the loaded module, and may ABEND if 16 bytes of data are not present. This situation can arise only if the entry point is an assembler (or other non-C) routine.

PORTABILITY

loadm is not portable. Be aware of system dependencies involving where load modules may be located and how module names are specified for your operating system.

IMPLEMENTATION

The implementation of **loadm** necessarily varies from operating system to operating system. Under MVS, modules to be loaded must reside in STEPLIB, a task library, or the system link list. Under CMS, modules to be loaded may reside in DYNAMC LOADLIB or in other locations defined by use of the **addsrch** routine.

loadm Dynamically Load a Load Module
(continued)

Under CICS, modules to be loaded must reside in a library in the DFHRPL concatenation and must be defined to CICS.

USAGE NOTES

addsrch does not verify the existence of a location, for example, DYNAMC LOADLIB. Because in some circumstances the logic of a program may not require that a location be searched, no verification is done until **loadm** cannot find a load module in any location defined earlier in the search order. **addsrch** fails only if its parameters are ill-formed.

If **loadm** determines that a location is inaccessible (for example, the LOADLIB does not exist), the location is marked unusable, and no attempt is made to search it again.

EXAMPLES

The use of **loadm** is illustrated by three examples. The first demonstrates the use of the command for a very simple situation without operating-system dependencies, while second and third examples are designed to run under MVS and CMS respectively.

The second example creates a dynamic load module that includes a table of pointers to functions which may be called dynamically from the calling load module. This example runs under MVS and has been designed to provide a framework that can be expanded upon in complete application.

The third example presents a hypothetical situation under CMS in which

1. the load module is created
2. the load module's location is added to the list of locations from which modules can be loaded (**addsrch**)
3. the load module is loaded (**loadm**)
4. the load module is deleted from the search order list (**delsrch**).

Example 1.1 simple case

```
#include <dynam.h>

int (*fp)();
   /* Load a load module named "ADD" and call it. */
loadm("ADD",&fp);
sum = (*fp)(1, 3);
.
.
.
```

Example 1.2 dynamic loading modules with multiple functions

Example 1.2 illustrates techniques for managing load modules containing multiple functions. This example includes illustrative MVS JCL, but the techniques illustrated in the example are also applicable to CMS.

loadm Dynamically Load a Load Module
(continued)

STEP I. Put the following declarations in a common header file and name it DYNTABLE:

```
struct funcdef {           /* structure definition for functions     */
   int (*func1)();
   int (*func2)();
                            /* More functions can go here.            */
};

typedef struct funcdef *fptrtable;  /* pointer to list of funcdefs    */
```

Make sure the header library containing DYNTABLE is allocated so that it will be included when you compile the following source code.

STEP II. Create the following C source file and name it DYNAMIC. This file will be compiled and linked to create a dynamic load module. The **_dynamn** function returns to its caller a structure of function pointers that can be used to call the individual functions of the load module.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <lcio.h>
#include <dynam.h>
#include "dyntable.h"

   /* _dynamn function that returns a list of function pointers which */
   /* can be used to invoke other functions in the load module        */
fptrtable _dynamn(void) {
      /* Initialize function pointer table.                           */
   static struct funcdef dyntab = {&func1, &func2 /* ... */};
      /* Return pointer to table with the new contents.               */
   return(&dyntab);
}

int func1() {
  printf("func1() was successfully dynamically called!!!\n");
  return(0);
}

int func2() {
  printf("func2() was successfully dynamically called!!!\n");
  return(0);
}
```

DYNAMIC must be compiled using the **sname** compiler option to override the default assignment of **_dynamn** as the section name. It may be compiled as re-entrant using the **rent** compiler option. (Note that JCL changes are required if **norent** compilation is needed.) During linking, the ENTRY=DYN (or DYNNK for **norent**) parameter must be specified.

loadm Dynamically Load a Load Module
(continued)

STEP III. Create the following C source file and name it DYNMLOAD. This code demonstrates how to dynamically load the DYNAMIC load module and how to call the functions pointed to by the **dyntab** table:

```
#include <stdio.h>
#include <string.h>
#include <dynam.h>
#include "dyntable.h"

fptrtable(*fpdyn)();  /* _dynamn function pointer prototype       */

int rc1, rc2;         /* return codes from calling func_1         */
                      /* and func_2                               */

fptrtable table;      /* table of function pointer and names      */
                      /* returned from _dynamn                    */
main()
{
      /* Load DYNAMIC.                                            */
   loadm("DYNAMIC", &fpdyn);

      /* Call _dynamn and return table of function pointers and   */
      /* name of load module.                                     */
   table = (*fpdyn)();

      /* Call func1 using function pointer from table.            */
   rc1 = (*table->func1)();
   printf("Dynamically called func1() with rc = %d \n", rc1);

      /* Call func2 using function pointer from table.            */
   rc2 = (*table->func2)();
   printf("Dynamically called func2() with rc = %d \n", rc2);

   unloadm(fpdyn);
   return;
}
```

STEP IV. Modify the following JCL, which is used to compile and link the DYNAMIC and DYNMLOAD source files and then execute the resulting load module. (Note that site-dependent job statements should be added.)

```
//*
//*   COMPILE and LINK module that uses a _dynamn routine that passes
//*   its caller back a list of function pointers which are used to
//*   invoke other functions in the load module
//*
//COMPDYNM EXEC LC370CL,ENTRY=DYN,PARM.C='RENT SNAME(DYNAM)'
//C.SYSLIN   DD DSN=userid.SASC.OBJ(DYNAMIC),DISP=OLD
//C.SYSIN    DD DSN=userid.SASC.SOURCE(DYNAMIC),DISP=SHR
```

loadm Dynamically Load a Load Module
(continued)

```
//*
//LKED.SYSLMOD  DD DSN=userid.SASC.LOAD(DYNAMIC),DISP=OLD
//*
//*
//*   COMPILE and LINK module that dynamically loads the DYNAMIC
//*   load module and calls functions within the load module
//*   using the structure of function pointers returned.
//*
//CLEMAIN EXEC LC370CLG
//C.SYSLIN   DD DSN=userid.SASC.OBJ(DYNMLOAD),DISP=OLD
//C.SYSIN    DD DSN=userid.SASC.SOURCE(DYNMLOAD),DISP=SHR
//*
//LKED.SYSLMOD  DD DSN=userid.SASC.LOAD(DYNMLOAD),DISP=OLD
//
```

Example 1.3 dynamic loading under CMS

A C source file, NEPTUNE C (listed below), is to be dynamically loaded and executed. The file contains the function **neptune**.

```
#include <stdio.h>

void neptune(char *p) {
   puts(p);
   return;
}
```

STEP I. Make the function **neptune** into a separate, loadable module by link-editing the TEXT file into a CMS LOADLIB file. The following steps are required:

1. Rename the function to **_dynamn**.

   ```
   #include <stdio.h>

   void _dynamn(char *p) {
      puts(p);
      return;
   }
   ```

   All C load modules (except the one that includes **main**, of course) must define one function named **_dynamn**.
2. Recompile NEPTUNE C, using the **sname** compiler option to override the default assignment of **_dynamn** as the sname. See Chapter 7, "Compiler Options," in the *SAS/C Compiler and Library User's Guide* for more information about the **sname** compiler option.
3. Link-edit the resulting NEPTUNE TEXT file using the CMS command LKED. The LIBE and NAME options of the LKED command can be used to specify the name of the output LOADLIB file and member name, respectively. For example,

   ```
   LKED NEPTUNE (LIBE DYNAMC NAME NEPTUNE
   ```

loadm Dynamically Load a Load Module
(continued)

The LKED command creates the file DYNAMC LOADLIB, containing the member NEPTUNE (assuming the LOADLIB did not already exist).

STEP II. The function **neptune** now exists in a form loadable by **loadm**. Invoke **loadm** to load the module as follows:

```
#include <dynam.h>
main()
{
   int (*fp)();              /* Declare a function pointer */
   loadm("NEPTUNE",&fp);     /* Load NEPTUNE               */
      if (fp) {              /* Check for errors           */
         (*fp)("Hello, Neptune, king of the C!");
         unloadm(fp);        /* Delete NEPTUNE             */
      }
      else
         puts("NEPTUNE failed to load.");
    exit(0);
}
```

STEP III. The previous step used the default search location DYNAMC LOADLIB (see **addsrch**); thus, no call to **addsrch** is required. If you use some other filename for the LOADLIB, specify it in a call to **addsrch** before invoking **loadm**. The following is an example:

```
#include <dynam.h>

main()
{
   SEARCH_P sp;                    /* Declare a SEARCH_P value  */
   int (*fp)();
      /* specify "NEWLIB LOADLIB *" */
   sp = addsrch(CMS_LDLB,"NEWLIB *","");
   loadm("NEPTUNE",&fp);
   if (fp) {
      (*fp)("Hello, Neptune, king of the C!");
      unloadm(fp);
   }
   else
      puts("NEPTUNE failed to load.");
   delsrch(sp);  /* remove NEWLIB LOADLIB from the search order */
   exit(0);
}
```

Since the NEPTUNE load module is relocatable, it can be loaded as a CMS nucleus extension by the NUCXLOAD command. Then the following calls cause NEPTUNE to be accessed from the nucleus extension:

```
sp = addsrch(CMS_NUCX,"","");
loadm("NEPTUNE",&fp);
```

loadm Dynamically Load a Load Module
(continued)

The function must exist as a nucleus extension before invoking `loadm`. This facility is useful, for example, in testing a single load module that you plan to replace in an existing LOADLIB.

unloadd Discard a Previously Loaded Data Module

SYNOPSIS

```
#include <dynam.h>;

void unloadd(MODULE mp);
```

DESCRIPTION

unloadd unloads the data module identified by the argument **mp**. If the module is no longer in use, it deletes the module from memory.

RETURN VALUE

None

ERRORS

If the argument to **unloadd** is invalid, a user 1211 ABEND is issued. Various other ABENDs, such as 1215 or 1216, may occur during **unloadd** if library areas used by dynamic loading have been overlaid.

CAUTION

If an attempt is made to use data in the unloaded module, the results are undefined but probably disastrous.

EXAMPLE

See **loadd**.

unloadm Discard a Previously Loaded Module

SYNOPSIS

```
#include <dynam.h>

void unloadm(__remote /* type */ (*fp)());
```

DESCRIPTION

unloadm unloads the executable module containing the function addressed by the argument **fp**. If the module is no longer in use, **unloadm** deletes it from memory. Note that **fp** may reference a function ruturning any valid type of data.

unloadm may be used to unload a module that has been built by **buildm**, but **unloadm** will not delete the module from memory.

RETURN VALUE

None

ERRORS

If the argument to **unloadm** is invalid, a user 1211 ABEND is issued. Various other ABENDs, such as 1215 or 1216, may occur during **unloadm** if library areas used by dynamic loading have been overlaid.

CAUTION

If an attempt is made to call a function in the unloaded module, the results are undefined but probably disastrous.

EXAMPLE

The following example, which is not system-specific, illustrates the general use of **unloadm**:

```
#include <dynam.h>

int (*fp)();
   /* Load a load module named "IEFBR14", */
   /* call it, and unload it.             */
loadm("iefbr14",&fp);
(*fp)();
unloadm(fp);
```

2 CMS Low-Level I/O Functions

2-1 Introduction
2-2 CMS Low-Level I/O Functions
2-5 XEDIT Low-Level I/O Functions
2-8 Related Utility Functions

Introduction

The library provides a set of functions that perform CMS disk file input and output operations. These functions are characterized as low-level I/O functions because they use the CMS file system directly. The library provides a second set of functions that can perform I/O operations on files in XEDIT storage. These functions are known as the XEDIT low-level I/O functions.

This chapter covers the low-level I/O functions as well as three utility functions that can be used with the I/O functions. Note that these functions are not portable.

The CMS low-level I/O functions are:

`cmsstate` verifies the existence of a CMS disk file

`cmsopen` opens a CMS disk file

`cmsread` reads a record from a CMS disk file

`cmswrite` writes a record to a CMS disk file

`cmspoint` changes the current record pointer for a CMS disk file

`cmsclose` closes a CMS disk file

`cmserase` erases a CMS disk file.

The low-level XEDIT I/O functions are:

`cmsxflst` verifies the existence of an XEDIT file

`cmsxflrd` reads a record from XEDIT storage

`cmsxflwr` writes a record to XEDIT storage

`cmsxflpt` moves the current line pointer in an XEDIT file.

The related utility functions are:

`cmspid` tokenizes a filename string in “cms” or “xed” style

`cmsdfind` finds a file identifier that matches a pattern

`cmsdnext` finds the next file identifier that matches a pattern.

CMS Low-Level I/O Functions

Descriptions of each CMS low-level I/O function follow.

SYNOPSIS

```
#include <cmsio.h>

int cmsstate(struct CMSFSCB *fscbp, struct CMSFST *fstp);
int cmsopen(struct CMSFSCB *fscbp);
int cmsread(struct CMSFSCB *fscbp);
int cmswrite(struct CMSFSCB *fscbp);
int cmspoint(struct CMSFSCB *fscbp);
int cmsclose(struct CMSFSCB *fscbp);
int cmserase(struct CMSFSCB *fscbp);
```

DESCRIPTION

The **cmsstate** function states or verifies the existence of a CMS disk file. The first argument to **cmsstate** is a pointer to a CMS File System Control Block (**CMSFSCB**) structure, and the second is a pointer to a CMS File Status Table (**CMSFST**) structure.

The remaining functions are as follows:

cmsopen	opens a CMS disk file.
cmsread	reads a record from a CMS disk file.
cmswrite	writes a record to a CMS disk file.
cmspoint	changes the current record pointer of a CMS disk file.
cmsclose	closes a CMS disk file.
cmserase	erases a CMS disk file.

The header file **<cmsio.h>** defines two structures for the CMS and XEDIT low-level I/O functions. The first structure maps a CMS FSCB and is defined as follows. (Note that extended FSCBs (FORM=E) are included.)

```
struct CMSFSCB {   /* CMSFSCB definition            */
   char comm[8];   /* file system command           */
   char fn[8];     /* filename                      */
   char ft[8];     /* filetype                      */
   char fm[2];     /* filemode                      */
   short itno;     /* relative record number        */
   char *buff;     /* address of r/w buffer         */
   int size;       /* length of buffer              */
   char fv;        /* recfm - C'F' or C'V'          */

   char flg;       /* flag byte                     */
   short noit;     /* number of records             */
   int nord;       /* number of bytes actually read */
   int aitn;       /* extended record number        */
   int anit;       /* extended number of records    */
   int wptr;       /* extended write pointer        */
   int rptr;       /* extended read pointer         */
};
```

The second structure maps a CMS FST and is defined as follows. (Again, note that the FORM=E fields are included.)

```
struct CMSFST {
   char fname[8];  /* filename                       */
   char ftype[8];  /* filetype                       */
   short datew;    /* date last written - MMDD       */
   short timew;    /* time last written - HHMM       */
   short wrpnt;    /* write pointer - item number    */
   short rdpnt;    /* read pointer - item number     */
   char fmode[2];  /* filemode - letter and number   */
   short recct;    /* number of logical records      */
   short fclpt;    /* first chain link pointer       */
   char recfm;     /* record format - F or V         */
   char flags;     /* fST flag byte (read/write)     */
   int lrecl;      /* logical record length          */
   short blkcnt;   /* number of 800 byte blocks      */
   short yearw;    /* year last written              */
   int fop;        /* alternate file origin pointer  */
   int adbc;       /* alt number of data blocks      */
   int aic;        /* alternate item count           */
   char nlvl;      /* number of ptr block levels     */
   char ptrsz;     /* length of a pointer element    */
   char adati[6];  /* alt date/time (YYMMDDHHMMSS)   */
   int _;
};
```

RETURN VALUE

All of the functions return the return code from their associated CMS macro. If the return code from FSSTATE is 0, **cmsstate** copies the information from the CMS FST to the **CMSFST** structure pointed to by **fstp**.

IMPLEMENTATION

Each function invokes the associated CMS macro. All of the functions expect an extended format (FORM=E) FSCB. The following table lists each function and the CMS macro it executes:

Function	CMS Macro
cmsstate	FSSTATE
cmsopen	FSOPEN
cmsread	FSREAD
cmswrite	FSWRITE
cmsclose	FSCLOSE
cmspoint	FSPOINT
cmserase	FSERASE

Refer to the appropriate IBM documentation for information about the data associated with the FSCB and FST and for information about these CMS macros.

EXAMPLE

The following example illustrates an interesting, if none too useful, change from the CMS TYPE command. This program reads a file backward and types each record to **stdout**.

```
#include <cmsio.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

main(int argc, char **argv)
{
   struct CMSFSCB fscb;
   struct CMSFST fst;
   register int rc;
   register char *buffer;

      /* Perform error checking as necessary.                    */
   if (argc < 2) {
      puts("Missing fileid");
      exit(4);
   }

   if (argc > 2) {
      puts("Extraneous parameters");
      exit(4);
   }

      /* Call cmspid to tokenize the fileid.                     */
   memset((void *) &fscb,'\0',sizeof(fscb));
   rc = cmspid(argv[1],&fscb);
   if (rc != 0) {
      printf("Fileid \"%s\" is invalid",argv[1]);
      exit(4);
   }

      /* Call cmsstate to get the file characteristics.          */
   rc = cmsstate(&fscb,&fst);
   if (rc != 0) {
      if (rc == 28)
         printf("File \"%s\" not found.\n",argv[1]);
      else
         printf("Error occurred while issuing FSSTATE.  RC=%d\n",rc);
      exit(rc);
   }

   buffer = malloc(fst.lrecl + 1);
   if (!buffer)
      exit(8);
```

```
      /* Fill in the required FSCB fields.                          */
   fscb.buff = buffer;
   fscb.size = fst.lrecl;
   fscb.fv = fst.recfm;
   fscb.anit = 1;

      /* Set current record number to number of records in file.   */
   fscb.aitn = fst.aic;

      /* Read the file backward; type records to the terminal.     */
   while ((rc = cmsread(&fscb)) == 0 && fscb.aitn > 0) {
      buffer[fscb.nord]= '\0';
      puts(buffer);
      --fscb.aitn;    /* Decrement the current record number.    */
   }

      /* Check for error while reading.                            */
   if (rc > 0) {
      printf("Error occurred while reading \"%s\".
              RC=%d\n", argv[1],rc);
      exit(rc);
   }
   exit(0);
}
```

XEDIT Low-Level I/O Functions

Descriptions of each XEDIT low-level I/O function follow.

SYNOPSIS

```
#include <cmsio.h>

int cmsxflst(struct CMSFSCB *fscbp, struct CMSFST *fstp);
int cmsxflrd(struct CMSFSCB *fscbp);
int cmsxflwr(struct CMSFSCB *fscbp);
int cmsxflpt(struct CMSFSCB *fscbp);
```

DESCRIPTION

The **cmsxflst** function states or verifies the existence of an XEDIT file. The first argument to **cmsxflst** is a pointer to a **CMSFSCB** structure, and the second argument is a pointer to a **CMSFST** structure.

The remaining functions are as follows:

cmsxflrd reads a record from XEDIT storage.

cmsxflwr writes a record to XEDIT storage.

cmsxflpt moves the current line pointer in an XEDIT file.

A pointer to a **CMSFSCB** structure is the only argument for these functions.

Refer to the "CMS Low-Level I/O Functions" on page 2-1 for a description of these structures.

RETURN VALUE

All of the functions return the return code from their associated XEDIT subcommand. If the return code from DMSXFST is 0, **cmsstate** copies the information from the CMS FST to the **CMSFST** structure pointed to by **fstp**.

IMPLEMENTATION

Each function invokes the associated XEDIT subcommand. All of the functions expect an extended format (FORM=E) FSCB. The following table lists each function and the XEDIT subcommand it executes:

Function	XEDIT Subcommand
cmsxflst	DMSXFLST
cmsxflrd	DMSXFLRD
cmsxflwr	DMSXFLWR
cmsxflpt	DMSXFLPT

Refer to the appropriate IBM documentation for information about the data associated with the FSCB and FST and for information about these XEDIT subcommands.

MACROS

Each of the XEDIT low-level I/O functions has an associated macro that can be used to provide a more readable name. The macros are defined in **<cmsio.h>** and are listed as follows:

```
#define xedstate(fscb,fst) cmsxflst(fscb,fst)
#define xedread(fscb) cmsxflrd(fscb)
#define xedwrite(fscb) cmsxflwr(fscb)
#define xedpoint(fscb) cmsxflpt(fscb)
```

EXAMPLE

The following example shows a program that uses **cmsxflwr** to write a date and time string at the current line of a file in XEDIT. The example shows how a program can communicate with XEDIT via the **system** function (using the "XEDIT prefix) and the **cmsxflst** function.

Note that XEDIT and either EXEC2 or REXX must be active when this function is executed. This sort of interaction between XEDIT and a program via an EXEC processor is most appropriate in a program using the SUBCOM interface or a program that is a REXX function package.

The example invokes the EXTRACT subcommand to place the filename, filetype, and filemode of the file into EXEC2 or REXX variables. Then, it calls **execfetch** to fetch the values of these variables and put them into a CMSFSCB structure. Next, it uses the **time** and **ctime** functions to create a date and time string. Finally, **cmsxflwr** writes the string into the file at the current line.

The example assumes that the file has fixed format records and a logical record length of 80. In practice, this information can be obtained with a call to **cmsstat**, **cmsxflst**, or via the EXTRACT subcommand. For clarity, all error checking after the initial call to **system** has been omitted.

For more information on the **time**, **ctime**, or **system** functions, refer to Chapter 6, "Function Descriptions," in *SAS/C Library Reference, Third Edition, Volume 2, Release 6.00*. For more information on the **cmsshv** function, refer to Chapter 8, "The CMS REXX SAS/C Interface" on page 8-1 in this book.

```
#include <cmsio.h>
#include <lclib.h>
#include <cmsexec.h>
#include <lcstring.h>
#include <time.h>

main()
{
   time_t now;
   struct CMSFSCB fscb;
   char timestring[80];
   int s, rc;

   rc = system("xedit: extract /fname/ftype/fmode");

   if (rc != 0) {
      printf("Unable to determine current fileid.  ");
      switch (rc) {
         case SYS_TNAC:
            puts("XEDIT is not active.");
            break;
         case SYS_CUNK:
            puts("Not called from EXEC2 or REXX exec.");
            break;
         default:
            if (rc > 0)
               printf("EXTRACT subcommand returned %d\n", rc);
            else
               printf("System function returned %d\n", rc);
            break;
      }
   }
   exit(rc);
}

      /* Set all fields in CMSFSCB structure to zeros.    */
      /* Fetch fileid components.                         */
   memset((void *) &fscb,'\0',sizeof(fscb));
   cmsshv(SHV_FETCH_DIRECT,"FNAME.1",7,fscb.fn,8,&s);
   cmsshv(SHV_FETCH_DIRECT,"FTYPE.1",7,fscb.ft,8,&s);
   cmsshv(SHV_FETCH_DIRECT,"FMODE.1",7,fscb.fm,2,&s);

   fscb.fv = 'F';           /* RECFM F                  */
   fscb.size = 80;          /* LRECL 80                 */
   fscb.buff = timestring;  /* record buffer address    */
   fscb.aitn = 0;  /* A zero indicates the current line.  */
```

```
      /* Initialize record buffer.  Get the date and time */
      /* and copy to the record. Call cmsxflwr to write   */
      /* the record.                                      */
   memset(timestring,' ',80);
   now = time(NULL);
   memcpy(timestring,ctime(&now),24);
   cmsxflwr(&fscb);

   exit(0);
}
```

Related Utility Functions

The **cmspid**, **cmsdfind**, and **cmsdnext** functions are often used in connection with CMS low-level I/O tasks. **cmspid** separates a CMS or XEDIT style filename string into filename, filetype, and filemode. **cmsdfind** and **cmsdnext** can be used to search for a specific CMS file. A description of **cmspid** follows. (See *SAS/C Library Reference, Third Edition, Volume 1* for descriptions of **cmsdfind** and **comsdnext**.)

cmspid Tokenize a CMS Fileid

SYNOPSIS

```
#include <cmsio.h>

int cmspid(const char *name, struct CMSFSCB *fscbp);
```

DESCRIPTION

cmspid tokenizes the string pointed to by **name** and fills in the filename, filetype, and filemode in the CMSFSCB structure pointed to by **fscbp**. The string pointed to by **name** may be any filename in the **cms** or **xed** style.

RETURN VALUE

cmspid returns 0 if tokenizing is successful and fills in the appropriate fields in the CMSFSCB structure. If **name** cannot be tokenized or does not refer to a CMS or XEDIT file, -1 is returned.

CAUTION

A return code of 0 from **cmspid** does not imply that the file exists.

EXAMPLE

```
#include <cmsio.h>

struct CMSFSCB fscb;

rc = cmspid("cms:profile.exec.a",&fscb);
.
.
.
```

SEE ALSO

"CMS Low-Level I/O Functions" on page 2-2.
"XEDIT Low-Level I/O Functions" on page 2-5.

3 MVS Low-Level I/O Functions

3-1 Introduction
3-2 DCBs and DCB Exit Routines
3-3 Direct BSAM Interface Functions
3-9 The Record-Oriented BSAM Interface
3-9 Controlling Buffering Using the Record Interface
3-10 Using osseek and ostell
3-10 Handling Spanned Records
3-11 Processing Concatenations with Unlike Attributes
3-11 BSAM Record-Oriented Interface Functions
3-16 The osdynalloc Function

Introduction

This chapter discusses functions that can be used for low level access to MVS sequential data sets. Three sets of functions are provided, two of which perform low-level I/O, and the third of which performs file allocation. There are two I/O interfaces: a full-function interface to BSAM and an interface similar to QSAM. The QSAM-like interface is record-oriented rather than block-oriented, and so is easier to use in many situations. Both of these interfaces access files by DDname, not by data set name. An additional function, **`osdynalloc`**, is provided to interface to the MVS dynamic allocation facility. This function can be used to allocate a data set to a DDname, or to obtain information about existing allocations. These functions are characterized as low level because they use the MVS access method and supervisor services directly.

This chapter covers these low-level I/O functions. Note that these functions are not portable. Though these functions are intended for use under MVS, they should function correctly under CMS, subject to the limitations of CMS BSAM simulation. MVS low-level I/O can be used in the minimal SPE environment as well as with the full run-time library.

MVS low-level I/O is supplied in source form, so it can be easily modified to support other access methods or unusual file types. See members L$UBSAM, L$UDCB, and L$UOSIO in SASC.SOURCE (MVS) or LSU MACLIB (CMS).

The direct BSAM interface functions are as follows:

`osbclose`	closes a BSAM DCB, and optionally frees the DCB.
`osbdcb`	allocates and initializes a BSAM DCB.
`osbldl`	returns location information for PDS members.
`osbopen`	opens a BSAM DCB.
`osbopenj`	opens a BSAM DCB using a TYPE=J OPEN macro.
`oscheck`	checks a read or write for errors.
`osfind`	finds a PDS member.
`osfindc`	positions to a PDS member using BLDL data.
`osnote`	returns the current file position.
`ospoint`	repositions a file.
`osread`	reads a block from a file.

ostclose	temporarily closes a BSAM DCB using a TYPE=T CLOSE macro.
osstow	updates a PDS directory.
oswrite	writes a block to a file.

The record-oriented interface functions are as follows:

osclose	closes and frees a DCB opened by **osopen**.
osdcb	allocates and initializes a DCB for record access.
osflush	flushes pending I/O so the file can be repositioned.
osget	reads a record from a file.
osopen	opens a DCB for record access.
osopenj	opens a DCB for record access using a TYPE=J OPEN macro.
osseek	repositions a file.
osput	writes a record to a file.
ostell	returns the current file position.

The functions **osfind**, **ostclose**, and **osstow** from the BSAM interface can be used with files opened using the record interface. You can also initialize a DCB using **osdcb** and then process it entirely with the direct BSAM interface. The IBM publication *MVS/DFP Using Data Sets* (SC26-4749) contains additional information about BSAM.

There is only one dynamic allocation interface function, **osdynalloc**. This function offers a number of different subfunctions, including allocation, deallocation, concatenation, and retrieval of attributes of data sets. The IBM publication *MVS/ESA Application Development Guide: Authorized Assembler Language Programs* (GC28-1645) provides additional information about dynamic allocation.

DCBs and DCB Exit Routines

The BSAM interfaces require that you allocate and initialize a DCB (data control block) using **osbdcb** or **osdcb**. The address of the DCB is then passed to the other I/O routines as an argument. Many BSAM functions require you to extract or modify data in the DCB. The header file **<osio.h>** contains a C structure definition for the DCB defining all necessary fields and constants.

Advanced use of BSAM in assembler frequently requires the coding of DCB exit routines, which are routines called by BSAM during processing of the DCB. (For instance, the SYNAD exit is called during processing of I/O errors.) Both BSAM interfaces enable you to write DCB exit routines in C. When you do this, the library takes care of linkage details for you so that the exit routine is called as a normal C function and the full facilities of the SAS/C Library can be used. (For instance, an assembler DCB exit routine is always entered in 24-bit addressing mode. However, before calling a C exit routine, the library switches back to 31-bit addressing mode, for a program that runs in that mode, so that data allocated above the 16-megabyte line can be accessed.)

When you use BSAM in assembler, you specify DCB exits by creating an *exit list*. Each entry in the list contains an exit type code and an entry address. The list of exits is then accessed via the DCBEXLST field of the DCB. When you use the SAS/C BSAM interface, the process is similar but not identical. You create a list of exits in which each entry contains an exit type code and a function address. (The C exit list format is not the same as the assembler format because of the need to support 31-bit

addressing.) The list of exits is passed to **osbdcb** or **osdcb** as an argument. This routine then transforms the C exit list into a similar assembler exit list, modifies the DCB, and actually performs the OPEN. (Note that some entries in an exit list are used as data addresses, such as a JFCB buffer address, rather than as function addresses. A different field name is used for storing a data pointer rather than a function pointer in an exit list.

When an exit routine is entered in assembler, parameters such as the address of the DCB or an I/O status block are passed in registers 0 and 1. When a corresponding C exit routine is called, it is passed two parameters, the first of which is the value that would be passed to the assembler exit in register 1, and the second is the register 0 contents. All exit functions should be declared as returning **int**, and the value returned is returned to BSAM in register 15.

Note that a C SYNAD exit requires slightly different linkage. When a C SYNAD exit is called, the library issues the SYNADAF macro before passing control to the exit routine. The address of the message constructed by SYNADAF is passed as the first argument, and the address of the DECB (data event control block) is the second argument. If the DCB address is required, it can be extracted from the SYNADAF message.

The BSAM interface supports escape from a DCB exit routine using the **longjmp** function. However, control is returned to data management with a value of **0** in register 15 before the **longjmp** is allowed to complete.

Note: When a DCB exit routine is running, use of some system services may cause task interlocks or ABENDs. Dynamic allocation and open are examples of such services. This means that you should not open a file in a routine called from a DCB exit. Also, be careful performing I/O to **stdin**, **stdout**, or **stderr** from a DCB exit because an operating system OPEN will be issued for these files if they have not been previously used. Finally, when debugging a DCB exit with the source level debugger, use the **auto nolist** command to prevent debugger access to the program source because the debugger uses dynamic allocation to gain access to the program source.

Direct BSAM Interface Functions

Descriptions of each direct BSAM interface function follow.

SYNOPSIS

```
#include <osio.h>

DCB_t *osbdcb(exit_t exit_list);
int osbopen(DCB_t *dcbp, const char *option);
int osbopenj(DCB_t *dcbp, const char *option);
int osbldl(DCB_t *dcbp, void *bldl_data);
int osfind(DCB_t *dcbp, const char *member);
int osfindc(DCB_t *dcbp, unsigned TTRK);
void osbclose(DCB_t *dcbp, const char *option, int free);
void ostclose(DCB_t *dcbp, const char *option);
void osread(DECB_t decb, DCB_t *dcbp, void *buf, int length);
void oswrite(DECB_t decb, DCB_t *dcbp, const void *buf,
             int length);
int oscheck(DECB_t decb);
unsigned osnote(DCB_t *dcbp);
void ospoint(DCB_t *dcbp, unsigned blkaddr);
int osstow(DCB_t *dcbp, const void *data, char type);
```

DESCRIPTION

osbdcb

The **osbdcb** function builds and initializes a BSAM DCB, and returns its address. The DCB address is then passed to the other routines to control their operation. The DCB includes a 16-byte extension of which 12 bytes are used by the library and 4 bytes are available for your use. (The name of the available field is DCBUSER.)

The argument to **osbdcb**, **exit_list**, is of type **exit_t []**, and the return value is of type **DCB_t ***. Both of these types are defined in **<osio.h>**. The definition of **exit_t** is as follows:

```
enum _e_exit {INACTIVE, INHDR, OUTHDR, INTLR, OUTTLR, OPEN, EOV, JFCB,
         USER_TOTAL=10, BLK_COUNT, DEFER_INTLR, DEFER_NONSTD_INTLR,
         FCB=16, ABEND, JFCBE=21, TAPE_MOUNT=23, SECURITY=24,
         LAST=128, SYNAD=256};              /* exit type            */

typedef __remote int (*_e_exit_fp)(void *, void *);
                                   /* exit function type             */
typedef struct _e_exit_list {      /* exit list entry definition     */
   unsigned exit_code;             /* actually an enum _e_exit,      */
                                   /* possibly with LAST bit set     */
   union {
      _e_exit_fp exit_addr;        /* exit function address          */
      void *area_addr;             /* pseudo-exit (e.g., JFCB) address*/
   };
} exit_t;
```

Consult IBM publication *MVS/DFP Using Data Sets* (SC26-4749) for information on the functions of the individual exits described by the codes above. Note that the last entry in the list must have the LAST bit set in its **exit_code** field. If only the LAST bit is set, the entry is considered inactive and ignored. You should specify an **exit_code** similar to (LAST | ABEND) if the last entry actually defines an exit.

If you need to pass a data address in the exit list, use the field name **data_addr** rather than **area_addr** to store it. Note that data addresses cannot be supplied as initial values for an exit list.

Note that the **exit_list** is converted by **osbdcb** into an assembler format exit list whose address is stored in DCBEXLST of the returned DCB. Modifications to the list passed to **osbdcb** after the call do not update the DCBEXLST value and, therefore, have no effect. Also note that if no exits are required, an argument of **0** should be passed to **osbdcb**.

The definition of **DCB_t** is too long to reprint here. It was generated from the DFP version 2 IHADCB DSECT using the DSECT2C utility. In addition to all the usual DCB symbols, the symbol DCBLRC_X has been defined as the DCBLRECL code stored to indicate LRECL=X.

Note: The definition of **DCB_t** requires the use of the compiler option **bitfield**, with a default allocation unit of **char**, in order to compile correctly.

osfind

The **osfind** function is called to issue the FIND macro for a DCB, to position the file to the start of a member. The member name is passed as a null-terminated string, in either upper- or lowercase. The value returned by **osfind** is the same as the return code from the FIND macro.

osbldl

The **osbldl** function can be used to locate PDS members more efficiently than with the **osfind** function. **osbldl** can look up more than one member at once.

The **osbldl** function is called to issue the MVS BLDL SVC to locate one or more members of a PDS. The **dcbp** argument is a pointer to the DCB for the PDS. The **bldl_data** argument is an area of storage defining the number and names of the members to be located. See IBM's *DFP Macro Instructions for Data Sets* for information on the format of the BLDL input data. **bldl_data** should be allocated below the 16 megabyte line. The value returned by **osbldl** is the **R15** value returned by the BLDL SVC.

If you are using the record-oriented BSAM interface, you may need to use **osflush** before calling **osfind** to quiesce any outstanding I/O.

osbopen, osbopenj

The functions **osbopen** and **osbopenj** are called to open a DCB using either the normal OPEN macro or an OPEN with TYPE=J. The **option** argument is a character string that should specify a valid OPEN macro keyword such as "input", "inout", or "updat". The string may be either upper- or lowercase. An invalid **option** is treated as "input". **osbopen** and **osbopenj** return **0** if successful or nonzero if unsuccessful.

osfindc

The **osfindc** function can be used to locate PDS members more efficiently than with the **osfind** function. **osfindc** can locate a member without having to search the PDS directory.

The **osfindc** function is called to position to a PDS member using data returned by the **osbldl** function by issuing the MVS FIND C macro. The **dcbp** argument is a pointer to the DCB for the PDS. The **TTRK** argument is the TTRK value returned by **osbldl** for the required member. The **osfindc** function returns the value stored in register 15 by the FIND C macro.

Using the record-oriented BSAM interface, you may need to use **osflush** before calling **osfind** to quiesce any outstanding I/O.

osbclose, ostclose

The functions **osbclose** and **ostclose** are called to close a BSAM DCB permanently (**osbclose**) or temporarily (**ostclose**). The **option** argument is a character string that should specify a valid CLOSE macro keyword such as "leave" or "reread". The string may be either upper- or lowercase. An invalid **option** is treated as "disp". The **free** argument of **osbclose** specifies whether the DCB should be freed after the close: **0** leaves the DCB allocated and nonzero frees it. If the DCB is freed, a FREEPOOL macro also is issued to release any buffers allocated by OPEN. Note that **osbclose** can process a DCB that has already been closed or never opened. This is useful for freeing a DCB that could not be opened.

Note: Files processed via MVS low-level I/O are not closed automatically by the library at program termination. This can cause a C03 ABEND for a program that opens one or more DCBs and then calls **exit**. You can use the

atexit library function to define a cleanup routine that closes all open DCBs to avoid this problem.

osread, oswrite

The functions **osread** and **oswrite** are called to read or write a block of data using a BSAM DCB. You must pass a DECB to control the operation. (This control block is later passed to **oscheck** to wait for the I/O to complete.) The data type **DECB_t** is defined by the library (as **unsigned [5]**) as the type of a DECB. You must also pass **osread** and **oswrite** the address of the buffer containing the data to be read or written and the length to read or write. The length is meaningful only for RECFM=U records; you can specify a length of **0** to use the value in DCBBLKSI. (A length of **0** is equivalent to the assembler specification 'S'.) No value is returned by **osread** or **oswrite** because you need to call **oscheck** to determine whether an I/O operation was successful. Note that the buffer passed to **osread** or **oswrite** must be allocated below the 16-megabyte line in MVS/XA. One way to assure this is to define the buffer areas as **auto** because **auto** variables always are allocated below the line.

oscheck

oscheck is called to wait for a READ or WRITE to complete and to determine whether the I/O was successful. The argument to **oscheck** is the address of the DECB for the operation to check. The value returned by **oscheck** is **0** for a successful read or write, **-1** if the end of the file was detected, and **-2** if an I/O error occurred (that is, if the SYNAD exit was called) or if some other condition occurred that caused the DCB to be closed. Note that information on the length of an input block is not returned. See IBM's *Using Data Sets* for information on determining this for various record formats.

osnote, ospoint

The **osnote** and **ospoint** functions are used to query or modify the file position for a BSAM file. For a sequential file, you must set the DCBMRPT1 and DCBMRPT2 bits in the DCB before the file is opened to use these functions. **osnote** returns the value returned by the NOTE macro in register 1. This is a block number for a tape file or a TTRz value for a disk file.

osstow

The **osstow** function is used to update a PDS directory by issuing the STOW macro. The arguments to **osstow** must be located in storage below the 16Mb line. To ensure this, the arguments should be defined as auto variables or if they are external or static variables the RENT compiler option should be specified. The **type** argument is a single character specifying the operation to be performed, such as 'A' to add a directory entry or 'D' to delete one. The format of the **data** argument varies according to the type of request, as described in IBM's *Using Data Sets*. Note that a member name contained in the **data** area should not be null-terminated and will *not* be translated to uppercase. The return value from **osstow** is the same as the register 15 return code from the STOW macro.

RETURN VALUE

Return values are described in the section above on a function-by-function basis.

IMPLEMENTATION

With the exception of **osbdcb**, each function invokes the associated BSAM macro. Refer to IBM's *Using Data Sets* and the *Macro Instructions for Data Sets* (IBM publication SC26-4747) for more information on individual macros.

EXAMPLE

This example copies the DDname INPUT to the DDname OUTPUT a block at a time. Only record formats F, V, and VB are handled to avoid code to determine block length. A DCB ABEND exit is provided to intercept B37 and similar ABENDs and terminate execution cleanly in this example.

Note that this example uses several other unique features of the compiler, such as the inline SVC interface and anonymous unions.

```
#include <osio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <getmain.h>

static int abend_exit(void *reg1, void *reg0);

        /* Register 1 argument to DCB ABEND exit. */
struct abend_info {
   unsigned abend_code: 12;
   unsigned :4;
   unsigned char return_code;
   union {
      struct {
         unsigned :4;
         unsigned recover: 1;
         unsigned ignore: 1;
         unsigned delay: 1;
         unsigned :1;
      }ok_to;
      char action;
   };
   DCB_t *dcbp;
   void *O_C_EOV_workarea;
   void *recovery_work_area;
};

int full;     /* Set to 1 if "file full" ABEND occurs. */

main()
{
   DCB_t *input, *output;
   exit_t out_exlst[1] = {LAST | ABEND, &abend_exit };
   DECB_t input_DECB, output_DECB;
   char *buf;
   int count = 0;
   int err;

   input = osbdcb(0);
   memcpy(input->DCBDDNAM, "INPUT   ", 8);
```

```
   if (osbopen(input, "input")) {
      puts("Input open failed.");
      exit(16);
   }
   output = osbdcb(out_exlst);

      /* Copy output file characteristics from input. */
   output->DCBRECFM = input->DCBRECFM;
   output->DCBLRECL = input->DCBLRECL;
   output->DCBBLKSI = input->DCBBLKSI;
   memcpy(output->DCBDDNAM, "OUTPUT  ", 8);
   if (osbopen(output, "output")) {
      puts("Output open failed.");
      osbclose(input, "", 1);
      exit(16);
   }
   buf = (char *) GETMAIN_U(input->DCBBLKSI, 0, LOC_BELOW);

      /* Allocate buffer below the 16 megabyte line. */
   for (;;) {
      osread(input_DECB, input, buf, 0);
      if ((err = oscheck(input_DECB)) != 0) {
         if (err != -1) puts("Input error.");
         break;
      }
      oswrite(output_DECB, output, buf, 0);
      if (oscheck(output_DECB) != 0) {
         if (full) puts("Output file full.");
         else puts("Output error.");
         break;
      }
      ++count;
   }

   printf("%d blocks copied.", count);
   FREEMAIN(buf, input->DCBBLKSI, 0, UNCOND);
   osbclose(output, "", 1);
   osbclose(input, "", 1);
   return 0;
}

   /* reg0 is undefined and unused for this exit.           */
static int abend_exit(void *reg1, void *reg0)
{
   struct abend_info *info;

   info = (struct abend_info *) reg1;
   if ((info->abend_code == 0xb37 ||
        info->abend_code == 0xd37 ||
        info->abend_code == 0xe37) && info->ok_to.ignore)
        /* if ignorable file full condition */
   {
      full = 1;
      info->action = 4;      /* Tell BSAM to ignore ABEND.   */
   }
```

```
    else
       info->action = 0;      /* Let any other ABEND proceed. */
    return 0;
}
```

The Record-Oriented BSAM Interface

The record-oriented BSAM interface is similar to QSAM in many ways, but the use of BSAM permits some useful non-QSAM functionality.

QSAM features supported by the BSAM record interface include the following:

- Access to data is a record at a time rather than a block at a time. Determination of input record length is handled automatically.
- Overlapping of I/O with CPU processing is handled automatically. The number of channel programs to issue and buffers to use is under program control.
- Fairly easy support is provided for processing concatenated sequential files with unlike attributes.

Differences between QSAM and the BSAM record interface include the following:

- The record interface allows the use of **`osfind`**, **`osstow`**, and **`ostclose`** to perform member-by-member PDS processing.
- Open modes of INOUT and OUTIN not supported by QSAM can be used.
- The **`osseek`** and **`ostell`** routines allow a file to be repositioned, which QSAM does not support.
- The processing of spanned records using the BSAM record interface is segment-oriented rather than record-oriented. Spanned record applications are harder to write with BSAM than they are with QSAM but still much easier than doing deblocking “by hand”.

The BSAM record interface also includes an **`osdcb`** routine, which can be used to define a DCB more conveniently than **`osbdcb`** by using a string of keyword parameters similar to the operands of the assembler DCB macro. Although this interface is more convenient than **`osbdcb`**, additional processing is required to parse the keyword string. This processing can be avoided by using **`osbdcb`** and explicit code to modify the DCB after it is allocated.

Controlling Buffering Using the Record Interface

The two DCB parameters that control buffering and overlapping of I/O operations are BUFNO and NCP. If these are not specified by the user when a file is opened, default values are assumed. Following are the four different styles of buffering supported:

1. BUFNO**`=1`** and NCP**`=1`**
   In this mode, no overlapping of I/O occurs. This mode is most useful for applications that use **`osseek`** heavily.
2. BUFNO**`=2`** and NCP**`=1`**
   In this mode, only one I/O operation is active, but a new operation usually starts immediately on completion of the previous request. This enables I/O and program processing to overlap and supports modest use of **`osseek`** and **`ostell`**.

3. BUFNO=*n* and NCP=*n*−1
   This mode provides for the maximum overlapping of I/O operations and is recommended for maximum efficiency. However, **osseek** and **ostell** are difficult or impossible to use.
4. BUFNO=*n* and NCP=*n*
   This mode is used with open mode "updat". **osseek** and **ostell** can be used only if *n* is **1**.

Using osseek and ostell

osseek and **ostell** are similar to the standard SAS/C Library's **fseek** and **ftell** functions. Unlike **osnote** and **ospoint**, they use position indicators (of type **ospos_t**) that identify a particular record, rather than a particular block, of the file. As part of its processing, **ostell** calls **osnote** to issue the NOTE macro, and, similarly, **osseek** calls **ospoint** to issue the POINT macro.

Unfortunately, a BSAM restriction requires that no I/O operations be active when a NOTE or POINT is issued. This means that when **ostell** or **osseek** is called, all reads or writes have to be checked, and the advantages of overlapping I/O are lost. There is another, more subtle problem as well.

Suppose the program has just called **osget** to obtain a record and now wants to call **ostell** to find the position of the record. **osget** may have started to read one or more additional blocks. After these read operations have been checked, a call to **osnote** returns the address of the last block read, which is not the same as the address of the block containing the record most recently passed to the program, the one whose address was requested. This problem means that **ostell** cannot usually be used meaningfully if the NCP value is greater than **1**.

To allow the use of **ostell** with BUFNO=**2**, which allows I/O to overlap with processing, the record interface supports a special mode of operation called "autonote" mode. In this mode, after every block is read or written, the library calls **osnote** and saves the block address for later use when the program calls **ostell**. Note that this requires that a NOTE be issued for every block, whether or not the block address is required. For this reason, autonote mode should be used only for applications in which the record address is used frequently or in which the efficiency gains from double buffering outweigh the loss from overuse of NOTE.

Autonote mode is set via an argument to **osopen** and cannot be changed once a DCB is opened.

Handling Spanned Records

The BSAM record interface mode is *locate* on input and *move* on output. That is, the input routine stores a pointer to the input data (within the BSAM buffer), and the output routine copies data to the BSAM buffer. Spanned records on input are difficult to deal with in locate mode because of the need to allocate a buffer to hold an entire record, when the length of the record is unknown, until all segments have been read. To bypass this problem, the SAS/C record interface is segment-oriented rather than record-oriented for spanned records. If segment consolidation is required, it can be performed by the application program, which frequently has information unavailable to the library about expected record lengths.

When the **osget** routine is called, it stores the address and length of the input record for nonspanned records. For spanned records, it stores the segment address and length. If the segment is the last (or only) segment of a record, the length is stored normally; for other segments, the negative of the length is stored. An application that needs to process an entire record at a time can call **osget** to read a segment at a time, stopping when a positive length is returned, indicating the end of the record.

Processing of spanned records is similar on output. When **osput** is called, one argument is the length of the record to write. If the length is negative, this indicates that the data are a record segment and that additional segments are to be expected. The final segment is indicated via a positive length. When record segments are passed to **osput**, the segments do not need to be physical segments because **osput** reblocks and resegments as necessary to match the output file attributes. This enables records to be copied easily from one record format to another without having to code elaborate logic to despan and respan records.

Processing Concatenations with Unlike Attributes

One reason for using a low-level I/O interface as opposed to a high-level interface such as standard C I/O is the ability to process concatenated files with different attributes, such as a tape file concatenated to a disk file or several files with different record lengths. Processing like this is easy with the direct BSAM interface. It is also supported via the record interface, but a little more work is necessary to enable the interface to rebuild its buffer pool and restart I/O when processing switches from one file to the next.

A brief summary of how to process unlike concatenations follows; for a more detailed description, see IBM publication *MVS/DFP Using Data Sets* (SC26-4749).

A program that supports concatenation of unlike data sets in assembler sets the DCBOFPPC bit of the DCBOFLGS field of the DCB. When the end of one of the concatenated data sets is reached, BSAM closes and reopens the DCB and frees and reallocates buffers for the file. Each time the DCB is closed and reopened, the DCB open exit is called by BSAM to inform the program that the file attributes may have changed and to give the program an opportunity to extract the new attributes from the DCB.

When the C record interface is used, the same DCB bit is used to request unlike concatenation support from BSAM. You must provide a C open exit, and the open exit must call the **osflush** routine to inform the record interface that the buffer pool has been freed and reallocated. Also, you must specify NCP=1 to allow the library to successfully restart I/O interrupted by the end of a concatenated file.

BSAM Record-Oriented Interface Functions

Descriptions of each BSAM record-oriented interface function follow.

SYNOPSIS

```
#include <osio.h>

DCB_t *osdcb(const char *ddn, const char *keywords,
             exit_t exit_list, char **errp);
int osopen(DCB_t *dcbp, const char *option, int autonote);
int osopenj(DCB_t *dcbp, const char *option, int autonote);
int osclose(DCB_t *dcbp, const char *option);
int osget(DCB_t *dcbp, void **bufp, int *lenp);
int osput(DCB_t *dcbp, const void *buf, int len);
int osflush(DCB_t *dcbp, int func);
int ostell(DCB_t *dcbp, ospos_t *posp);
int osseek(DCB_t *dcbp, ospos_t pos);
```

DESCRIPTION

osdcb

The **osdcb** function builds and initializes a BSAM DCB and returns its address. It is somewhat easier to use in complicated applications than **osbdcb** but has more overhead. Either **osdcb** or **osbdcb** can be used to build a DCB to be processed by either BSAM interface. As with the DCB built by **osbdcb**, the DCB includes a 16-byte extension area for library and user use.

The **ddn** argument to **osdcb** is the DDname of the file to open. The DDname should be null-terminated and can be in either upper- or lowercase. A **ddn** value of **0** can be specified if the DDname is not known when the DCB is built. (In this case, the DDname must be stored in the DCBDDNAM field by the program before the DCB is opened.)

The **keywords** argument to **osdcb** is a null-terminated string containing DCB macro keywords, such as "dsorg=po,bufno=5". The supported keywords are DSORG, RECFM, LRECL, BLKSIZE, OPTCD, NCP and BUFNO. Keywords and their values can be specified in either upper- or lowercase. If several keywords are specified, they can be separated by blanks or commas.

The **exit_list** argument of **osdcb** is the same in all respects as the **exit_list** argument of **osbdcb**. See the description of that function for details.

The **errp** argument of **osdcb** is used to control the processing of errors in the **keywords** string. If **errp** is **0** and there is an error in the **keywords** string, a library warning message is written to **stderr**. If **errp** is not **0**, it should address a **char *** variable, in which is stored the address of the invalid keyword. This pointer can be used by the program to send a diagnostic or correct the error.

osdcb returns the address of the new DCB or **0** if no DCB was created because of an error in the **keywords** string.

osopen, osopenj

osopen and **osopenj** open a BSAM DCB for record-oriented access using a normal OPEN macro or an OPEN TYPE=J respectively. The **option** argument is a character string that should specify a valid OPEN macro keyword such as "input", "inout" or "updat". The string can be either upper- or lowercase. An invalid **option** is treated as "input". The **autonote** argument is an integer specifying whether the NOTE macro should be issued automatically after a READ or WRITE. A value of **0** means that NOTE is not issued automatically, and a nonzero argument means it is issued automatically. See "Using osseek and ostell" on page 3-10 for more information on **autonote**.

When you use the record interface, RECFM=D data sets can be processed only with BUFOFF=L. (You do not need to set this option yourself; it is set automatically by the library.)

osopen and **osopenj** return **0** if successful or nonzero if unsuccessful.

osclose

osclose is called to close and free a DCB opened using **osopen** or **osopenj**. All buffers also are freed at the same time. The **option** argument is a character string that should specify a valid CLOSE macro keyword such as "leave" or "reread". The string can be either upper- or lowercase. An invalid **option** is treated as "disp". **osclose** returns **0** if the DCB is closed successfully or nonzero if any problems occur. Even if the return code is nonzero, the DCB will have been freed. (A nonzero return code generally indicates a problem

flushing buffers.) Note that **osclose** can be used to free the storage for a DCB that failed to open.

osget

The **osget** function is called to read a record or record segment and return its address and length. The **bufp** argument addresses a **void *** variable in which the address of the record or record segment will be stored. The value stored always addresses the data, rather than the record prefix for V-format data. The **lenp** argument addresses an **int** variable in which to store the record or segment length. For a spanned data set, the negative of the segment length is stored except for the last or only segment of a record.

The return value from **osget** is **0** for normal completion, **-1** for the end of the file, or **-2** for some other error. See **oscheck** for more details on the return code meaning.

osput

The **osput** function is called to write a record or record segment. The **buf** argument is a **void *** pointer addressing the record or record segment to be written. The record should contain only data; any record prefix needed is built by the library. The **len** argument is the length of the record or record segment to be written. For a spanned record segment, **len** should be positive for the last (or only) segment of a record or, in any other case, the negative of the segment length. If the record length does not match the DCB LRECL specification, the record is padded with nulls or truncated. (This is not considered an error.) A length of **0** can be used to terminate a spanned record without adding any additional data.

The return value from **osput** is **0** in the normal case, **-2** if an I/O error occurred, or **-3** if **osput** was called for a file opened for UPDAT to write more bytes than were returned by the previous **osget**.

osflush

The **osflush** function is called to terminate all I/O to a DCB so the DCB can be modified safely. For instance, you should call **osflush** for a DCB before you issue the BSP SVC to backspace the file. The **func** argument to **osflush** can take on one of four values: **QUIESCE**, **REPOS**, **SYNCH**, or **CONCAT**. These macros are defined in **<osio.h>**.

A **func** value of **QUIESCE** to **osflush** informs the library that the position of the file will not be changed by processing after **osflush** completes. This enables the library to retain any blocks that are already read at the time **osflush** is called.

A **func** value of **REPOS** to **osflush** informs the library that the position of the file may change, or that for some other reason, all information about current file contents retained by the library should be purged. Specifying REPOS unnecessarily causes additional library processing the next time **osget** or **osput** is called for the DCB.

A **func** value of **SYNCH** to **osflush** specifies the same processing as **REPOS**, except that at the completion of all other processing, a CLOSE TYPE=T is issued to the DCB, thus writing an end-of-file mark and updating the directory for a PDS member.

A **func** value of **CONCAT** should be passed to **osflush** only for a call from an open exit for a concatenated input file. It handles resynchronization and reallocation of buffers, and it should not be used in any other circumstances.

The return value from **osflush** is normally **0**, but can be a negative value returned by **oscheck** if errors occur during processing.

ostell

The **ostell** function is called to store the current record address for a BSAM file. The argument **posp** is a pointer to an area of type **ospos_t**, in which the current record address should be stored. The definition of **ospos_t** is as follows:

```
typedef struct {
   unsigned _blkaddr;
   unsigned _recno;
} ospos_t;
```

The first field is the block address for the current block (as returned by **osnote**) and the second field is the current record number. See "Using osseek and ostell" on page 3-10 for more information on this function.

ostell returns **0** if successful or a nonzero value if it fails.

osseek

The **osseek** function is called to reposition a BSAM file. The argument **pos** is a value of type **ospos_t** defining the record to seek. It is not necessary to call **osflush** before calling **osseek**; **osseek** does this automatically. See "Using osseek and ostell" on page 3-10 for more information on this function.

osseek returns **0** if successful or a nonzero value if it fails. Note that an invalid argument to **osseek** may cause an ABEND or incorrect results later in processing rather than a nonzero return code.

RETURN VALUE

See the function descriptions here for return code details.

IMPLEMENTATION

These functions are implemented by the L$UOSIO module, which issues calls to the direct BSAM interface as necessary to perform I/O.

EXAMPLE

This example copies the contents of DDname INPUT to the DDname OUTPUT, and then uses the **osstow** function to update the PDS directory. The **osflush(REPOS)** function is used to force all output buffers to disk before updating the directory. Also, note the use of an open exit to define default attributes for the output data set.

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

static DCB_t *input, *output;
static int open_exit();

main()
{
   exit_t out_exlst[1] = {LAST | OPEN, &open_exit};
   char *rec;
   int length;
   int err;
   int count = 0;
   struct {
```

```
      char name[8];
      unsigned TTRC;
   } stow_data = {"MYMEMBER", 0};

   input = osdcb("input", "recfm=fb,lrecl=80,bufno=10", 0, 0);
   if (osopen(input, "input", 0)) {
      puts("Input open failed.");
      exit(16);
   }
   output = osdcb("output", "recfm=fb,lrecl=80,bufno=10,dsorg=po",
                  out_exlst, 0);
   if (osopen(output, "output", 0)) {
      puts("Output open failed.");
      exit(16);
   }

   for (;;) {
      err = osget(input, &rec, &length);
      if (err != 0) {
         if (err != -1) puts("Input error.");
         break;
      }
      err = osput(output, rec, length);
      if (err != 0) {
         puts("Output error.");
         break;
      }
      ++count;
   }

   if (err == 0) {                    /* if there were no errors   */
      err = osflush(output, REPOS);   /* Flush output buffers.     */
      if (err == 0)
            /* Add member to PDS directory. */
         err = osstow(output, &stow_data, 'A');
   }
   if ((input->DCBRECFM & (DCBRECU | DCBRECSB)) ==
       (DCBRECV | DCBRECSB))          /* if input file was spanned */
      printf("%d segments copied.\n", count);
   else
      printf("%d records copied.\n", count);
   osclose(output, "");
   osclose(input, "");
}

static int open_exit(reg1, reg0)
DCB_t *reg1;
void *reg0;
{
   /* If output file attributes unset, copy from input attributes. */
   if (reg1->DCBRECFM == 0) {
      reg1->DCBRECFM = input->DCBRECFM;
```

```
        reg1->DCBLRECL = input->DCBLRECL;
        reg1->DCBBLKSI = input->DCBBLKSI;
    }
    return 0;
}
```

The osdynalloc Function

The following is a description of the osdynalloc function.

SYNOPSIS

```
#include <os.h>

int osdynalloc(int action, char *keywords, char *libmsgbuf, ...);
```

DESCRIPTION

osdynalloc is used to invoke the MVS dynamic allocation SVC (SVC 99). You can use **osdynalloc** to allocate, free, concatenate or deconcatenate data sets, as well as to return information about existing allocations. MVS dynamic allocation is described in detail in the IBM publication *MVS/ESA Application Development Guide: Authorized Assembler Language Programs.* Unless you are already familiar with dynamic allocation, you should read the sections of this book discussing dynamic allocation to be aware of the restrictions and other complexities of this powerful service.

The **action** argument to **osdynalloc** specifies the required dynamic allocation action. The names of the actions (defined in **<os.h>**) are:

DYN_ALLOC	allocate a data set
DYN_FREE	free a data set or DDname
DYN_CONCAT	concatenate several DDnames
DYN_DECONCAT	deconcatenate a concatenated DDname
DYN_DDALLOC	allocate a data set by DDname
DYN_NOTINUSE	mark allocations not in use
DYN_INQUIRE	obtain information about current allocations

Note: The **DYN_DDALLOC** action is very specialized. See IBM documentation for information about the use of **DYN_DDALLOC** and how it differs from **DYN_ALLOC**.

The **keywords** argument is a pointer to a character string containing a list of keywords. The keywords specify parameters for the request, for instance the name of a data set to be allocated or the address of a variable in which to store a DDname. See below for some simple examples of keyword strings.

The **libmsgbuf** argument specifies the address of an array of characters (containing at least 128 bytes) in which any error message generated by the library is to be stored. If **libmsgbuf** is specified as **NULL**, the library writes any error messages to **stderr**, and the texts are not returned to the caller. Note that in SPE if **libmsgbuf** is specified as **NULL**, no messages will be written unless the **$WARNING** routine has been replaced, as described in the *SAS/C Compiler and Library User's Guide, Fourth Edition.* See "DIAGNOSTICS" later in this section for further information on error handling.

Additional arguments to **osdynalloc** may optionally be specified after the **libmsgbuf** argument. These arguments are used to complete the **keywords** string, as shown in the examples below.

Here are a few simple calls to **osdynalloc**, to show how keyword strings are assembled:

```
rc = osdynalloc(DYN_ALLOC,
      ''ddn=master,dsn=billing.master.data,disp=shr", NULL);
```

allocates the data set BILLING.MASTER.DATA with disposition SHR to the DDname MASTER.

```
rc = osdynalloc(DYN_FREE,
      ''ddn=master,disp=delete,reason=?", NULL, &reason);
```

frees the DDname MASTER with disposition DELETE, and stores the error reason code in the integer **reason**.

```
rc = osdynalloc(DYN_CONCAT,
      ''ddn=(syslib,syslib1,syslib2),perm", NULL);
```

concatenates the DDnames SYSLIB, SYSLIB1 and SYSLIB2, and marks the concatenation as permanent.

```
rc = osdynalloc(DYN_INQUIRE,
      ''ddn=master,retdsn=?,retdsorg=?,msgcount=?,errmsgs=?",
      libmsgbuf, dsnbuf, &dsorg, &number_msgs, &mvsmsgbuf);
```

finds the name of the data set allocated to the DDname MASTER and its organization, and stores the results in the variables **dsnbuf** and **dsorg**. If the request fails, any library message is stored in **libmsgbuf**, and the address of an MVS message buffer is stored in **mvsmsgbuf**. Also, the number of MVS messages is stored in **number_msgs**.

The **keywords** argument to **osdynalloc** is just a list of items separated by commas. Each item is identified by a keyword, like **''ddn''**, **''disp''** and **''perm''** in the examples above. The permitted keywords are determined by the particular action requested by **action**, and are described in tables appearing later in this description. Each keyword in the string is translated by **osdynalloc** into a single dynamic allocation text unit. (See *Authorized Assembler Language Programs* for further information on text units.) Additional keywords are accepted in calls to **osdynalloc**, regardless of **action**, such as **''reason''** and **''errmsgs''** in the examples above. These keywords correspond to fields in the SVC 99 request block, and generally request special processing options or deal with error-handling.

Syntactically, there are three kinds of **keywords** items, as follows:

- single value items, like **''disp=shr''**. These items have the form **"***keyword***=***value***"**.
- multiple value keywords, like **''ddn=(syslib,syslib2)''**. These items have the form **"***keyword***=(***value1***,***value2***, . . .)"**. If there is only one value, the parentheses can be left off.
- switch items, like **''dummy''**. These items have a keyword, but no value.

Any keyword value may be specified as the single character **''?''**. This indicates that the actual keyword value is present in the argument list as an additional argument. The order of any additional arguments is the same as the order of the corresponding keywords in the **keywords** string. Note that for multiple value keywords, each ? corresponds to a single value. That is, you could have a call like the following:

```
osdynalloc(DYN_ALLOC, "...volser=(?,?),...", NULL, "VOLUM1", "VOLUM2")
```

However, you cannot have a call like the following:

```
osdynalloc(DYN_ALLOC, "...volser=?,...", NULL, "(VOLUM1,VOLUM2)")
```

The keywords accepted by **osdynalloc** are classified into various sorts based on the type of the associated values:

String values
: are associated with keywords, such as **''member=name''**, that have a value which is a character string. An argument supplied using the **''?''** convention should have type **char ***. With a few exceptions, string values are automatically translated to upper-case.

Dsname values
: are associated with keywords, such as **''dsn=.project.c''**, that have a string value that is interpreted as a data set name. If the name begins with a period, the name is completed by prepending the user's TSO prefix or (in batch) the userid.

Reserved-word values
: are associated with keywords, such as **''disp=old''**, that have a string value which is limited to a prescribed set of values. Except for this limitation, they are treated as string values. The values permitted for particular keywords are the values permitted for the corresponding JCL parameter, unless stated otherwise below.

Integer values
: are associated with keywords, such as **''blksize=800''**, that have a value that is a decimal or hexadecimal string. An argument supplied using the **''?''** convention should have type **int**.

Pointer values
: are associated with keywords, such as **''retddn=0xf0328''**, that have a value that is a decimal or hexadecimal address, pointing to an area where information is to be returned. This must be the address of a variable of the appropriate type, either **int** for numeric information, or a **char** array for string information. In many cases, the values returned via pointers are encoded. For instance, a dsorg (data set organization) is returned as an integer with different bits set depending on the actual file organization. Information on interpreting these values is given in the IBM book cited above.

There are also keywords with specialized value representations (such as **''secmodel=''** and **''expdt=''**). These are discussed individually in the keyword tables below.

RETURN VALUE

osdynalloc returns **0** if it was successful. It returns a negative value if an error was detected by the C library before the dynamic allocation SVC could be invoked. In this case, a diagnostic message will have been stored in **libmsgbuf** if this address is not **NULL**. If SVC 99 is called and fails, **osdynalloc** returns the value returned in register 15 by SVC 99, as described in *Authorized Assembler Language Programs.* Additional information about the error can be obtained by use of keywords such as **''reason''**, **''inforeason''** and **''errmsgs''**, as described in the SVC 99 request block keyword table below.

CAUTIONS

Not all keywords listed below are supported by all versions of MVS. For instance, the **``path''** keyword is only supported when OpenEdition MVS is installed.

DIAGNOSTICS

osdynalloc is subject to three different sorts of errors.

The first sort is errors detected before calling SVC 99. Examples of such errors are unrecognized keywords or memory allocation failures. The SAS/C library generates diagnostics for these failures and either writes them to **stderr** or returns them to the user via the **libmsgbuf** argument. SVC 99 is not issued if one of these errors occurs.

The second sort is errors detected by SVC 99. Information about these errors is returned in the **osdynalloc** return code, and additional information can be obtained via keywords like **``reason''**. The library never diagnoses any of these errors. The **``errmsgs''** keyword may additionally be specified to request that one or more messages about a failure be returned to the caller. These messages are obtained by a call to the IBM routine IEFDB476. Note that these messages are frequently unsuitable for particular applications. (For example, the message for an out-of-disk-space situation exhorts the user to "**USE THE DELETE COMMAND TO DELETE UNNEEDED DATA SETS**", even if the program is not running under TSO.) Also note that the **``issuemsg''** keyword can be used to request that dynamic allocation issue these messages itself, using either the PUTLINE service or the WTO SVC.

The third sort of error is errors detected by IEFDB476. These errors are not diagnosed by the library or by MVS, and do not affect the **osdynalloc** return code. The IEFDB476 error codes can be accessed using the **``msgerror''** and **``msgreason''** keywords.

IMPLEMENTATION

osdynalloc is implemented by the L$UDYNA module, which is provided in source form. You can modify this module to add or delete keywords. You might wish to add keywords if functionality is added to SVC 99 by a new release of MVS. You might wish to delete keywords if you have a storage-constrained application which does not need to use some of the more obscure dynamic allocation options or keywords.

KEYWORD TABLES

Table 3.1 on page 3-20 shows all the keywords that can be used for any call to **osdynalloc**, regardless of which action is specified. These keywords are not translated to text units. Rather, they cause information to be stored in the SVC 99 request block (S99RB) or request block extension (S99RBX). Most options can be identified by more than one keyword. This is intended to assist you to easily locate the keywords you need. Short forms correspond to the field names defined in *Authorized Assembler Language Programs*, as well as longer names which may be more understandable or easier to recall.

Note that some options in this table can only be used by authorized programs, as described in the IBM manual.

Table 3.1 *SVC 99 Request Block Keywords*

Identifier	RB Field	Value	Description	Notes
condenq cnenq	S99CNENQ	none	Conditionally ENQ on TIOT	
cppl ecppl	S99ECPPL	void *	TSO CPPL address	
errmsgs errmsg ermsg emsgp	S99EMSGP	dyn_msgbuf**	Error message buffer return address	**(1)**
errorcode errorreason reasoncode reason error	S99ERROR	int *	Error reason code return address	
flags1 flag1 flg1	S99FLAG1	int	First S99RB flag byte	**(2)**
flags2 flag2 flg2	S99FLAG2	int	Second S99RB flag byte	**(2)**
freereason freeerror ercf	S99ERCF	int *	Message free error code	
gdglocate gdgnt	S99GDGNT	none	Always use the most recent GDG catalog information	
infoerror infoerr eerr	S99EERR	int *	Info retrieval error code return address	
infoinfo einfo	S99EINFO	int *	Info retrieval informational code return address	
inforeason infocode info	S99INFO	int *	SVC 99 informational reason code return address	
issuemsg sendmsg eimsg	S99EIMSG	none	Send SVC 99 messages before returning	
jobsysout jbsys	S99JBSYS	none	Treat SYSOUT as normal job output	

continued

Table 3.1 (*continued*)

Identifier	RB Field	Value	Description	Notes
key ekey emkey	S99EKEY	int	Storage key for SVC 99 messages	
mount	S99MOUNT	none	Allow mounting of volumes	
msgbelow lsto	S99LSTO	none	Allocate SVC 99 messages below the 16 meg line	
msgcount nmsgs enmsg nmsg	S99ENMSG	int *	Number of SVC 99 messages return address	
msgerror	none	int *	Message processing error code (returned in register 15 by IEFDB476)	
msgflags msgopts eopts	S99EOPTS	int	S99RBX option bits	**(2)**
msgreason erco	S99ERCO	int *	Message processing reason code return address	
msgseverity msglevel emsgsv msgsv emgsv	S99EMGSV	int	Minimum severity of SVC 99 messages	**(3)**
mustconvert oncnv	S99ONCNV	none	Use an existing allocation only if it is convertible	
noconvert nocnv	S99NOCNV	none	Do not convert an existing allocation	
noenq tionq	S99TIONQ	none	Do not ENQ on SYSZTIOT	
nomigrate nomig	S99NOMIG	none	Do not recall migrated data sets	
nomount nomnt	S99NOMNT	none	Do not mount volumes or consider offline devices	
nomsg msgl0	S99MSGL0	none	SVC 99 should not issue any messages	
noreserve nores	S99NORES	none	Do not reserve data sets	

continued

Table 3.1 (*continued*)

Identifier	RB Field	Value	Description	Notes
offline offln	S99OFFLN	none	Consider offline devices	
putlinerc wtorc wtprc ewrc	S99EWRC	int *	PUTLINE/WTO return code return address	
smsreason ersn	S99ERSN	int *	SMS reason code return address	
subpool esubp emsub subp	S99ESUBP	int	Subpool number for SVC 99 message allocation	
unitdevtype udevt	S99UDEVT	none	Interpret unit name as a DEVTYPE value	
waitdsn wtdsn	S99WTDSN	none	Wait for data sets	
waitunit wtunit wtunt	S99WTUNT	none	Wait for units	
waitvol wtvol	S99WTVOL	none	Wait for volumes	
wtp wto ewtp	S99EWTP	none	Send messages with WTP	

(1) The value returned is not the value stored in S99EMSGP. **`osdynalloc`** calls IEFDB476 to turn the S99EMSGP value into an array of message buffers, a pointer to which is returned to the user in the variable specified by the keyword. Each element of the array is of type **`dyn_msgbuf`**. (This type is defined in **`<os.h>`**.) The buffer should be released by a call to **`free`** after the messages have been processed.

If an error occurs when the library attempts to convert S99EMSGP to a message buffer array, **`(char *) -1`** is stored in the return area. The return value from **`osdynalloc`** is not affected.

(2) Use of these keywords is not recommended because they store an entire byte of flags. Use of the keywords for individual flags will result in more easily understandable programs. If you use one of these keywords, any flag bits turned on by previous keywords in the string will be lost. For instance, if you code the keywords **`nomount,flag1=0x40`**, the nomount bit will not be set.

(3) Valid values for this keyword are **`0`** for all messages, **`4`** for warning and error messages, and **`8`** for only error messages.

Tables 3.2 through 3.8 show, for each dynamic allocation action, the supported keywords and their characteristics. For each keyword, the table shows the JCL equivalent (if any), the corresponding SVC 99 text unit key (which can be used to locate additional information about the keyword in *Authorized Assembler Language Programs*), the expected C type and format of the keyword value (if any), and a brief description. For keywords whose values are restricted to a specific size, `char[n]` is shown as the type, where `n` is the array size needed to hold the largest permitted value. This includes space for a terminating null so that, for instance, DDnames are shown as `char[9]`, even though the DDname itself is limited to eight characters. For values shown as pointers to arrays, the array passed must be of exactly the size shown.

Most options can be identified by more than one keyword. This is intended to assist you in easily locating the keywords you need. Short forms are provided which correspond to the dynamic allocation key names, as well as longer names which may be more understandable or easier to recall.

The Format column of the table may contain one or more of the following values:

Dsname	The keyword value is a data set name, and may be specified with an initial period to request that the TSO prefix or userid be prepended.
Encoded	The stored value is encoded as an integer. See the IBM documentation, the *Authorized Assembler Language Programs*, book for a description of the encoding for a particular keyword.
Multiple	The keyword allows more than one value to be specified.
Optional	The keyword permits a value to be specified, but one is not required.
Res Word	The keyword value is a reserved word (for example, `''disp=''`) or a string of single-letter flags (for example, `''recfm=''`). The permitted values are described in the IBM documentation, *MVS/ESA JCL Reference.*

Table 3.2 *Dynamic Allocation Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
accode acode	ACCODE=	DALACODE	char[9]		ANSI tape accessibility code	
avgrec avgr	AVGREC=	DALAVGR	char *	Res Word	Average record size multiplier	
blocklen blklen blkln	SPACE=(len, . . .)	DALBLKLN	int		Average block length for space allocation	
blocksize blksize blksiz blksz dcbblksize dcbblksz	DCB=BLKSIZE=	DALBLKSZ	int		Maximum block size	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
bufalign bufaln bfaln dcbbfaln	DCB=BFALN=	DALBFALN	char *	Res Word	Buffer alignment	
bufin dcbbufin	DCB=BUFIN=	DALBUFIN	int		Number of initial TCAM input butters	
bufmax bufmx dcbbufmax dcbbufmx	DCB=BUFMAX=	DALBUFMX	int		Maximum number of TCAM buffers	
bufno dcbbufno	DCB=BUFNO=	DALBUFNO	int		Number of buffers to allocate	
bufoff bufof dcbbufoff dcbbufof	DCB=BUFOFF=	DALBUFOF	int		ANSI tape buffer prefix offset	**(1)**
bufout bufou dcbbufout dcbbufou	DCB=BUFOUT=	DALBUFOU	int		Number of initial TCAM output buffers	
bufsize bufsiz bufsz dcbbufsize dcbbufsz	DCB= BUFSIZE=	DALBUFSZ	int		TCAM buffer size	
buftech buftek bftek dcbbftek	DCB=BFTEK	DALBFTEK	char *	Res Word	Buffering technique	
burst	BURST=	DALBURST	char *	Res Word	3800 printer burst specification	
cdisp	DISP=(s,n,cd)	DALCDISP	char *	Res Word	Conditional (ABEND) disposition	
chars	CHARS=	DALCHARS	char[5]		3800 printer character arrangement table	
cntl	CNTL=	DALCNTL	char[27]		Reference a CNTL JCL statement	
convertible convert cnvrt	none	DALCNVRT	none		Make this allocation convertible	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
copies copys copy	COPIES=	DALCOPYS	int		Number of SYSOUT copies	
copyg	COPIES=(,(g,...))	DALCOPYG	int	Multiple	3800 printer copy groups	
cpri dcbcpri	DCB=CPRI=	DALCPRI	char *	Res Word	TCAM relative transmission priority	
cyl	SPACE=(CYL,...)	DALCYL	none		Allocate space in cylinders	
dataclass dataclas dacl	DATACLAS=	DALDACL	char[9]		SMS data class	
dcbddname dcbddn dcbdd	DCB=*.ddn	DALDCBDD	char[9]		Copy DCB information from DD statement	
dcbdsname dcbdsn dcbds	DCB=dsn	DALDCBDS	char[47]	Dsname	Copy DCB information from data set	
ddname ddnam ddn	none	DALDDNAM	char[9]		DDname to allocate	
defer unitdefer	UNIT=(,,DEFER)	DALDEFER	none		Defer mounting until open	
den dcbden	DCB=DEN=	DALDEN	char *	Res Word	Tape density	
dest destnode node suser	DEST=	DALSUSER	char[9]		SYSOUT destination node	
destuser userid usrid user	DEST=(n,user)	DALUSRID	char[9]		SYSOUT destination userid	
diagnstrace diagnose diagns diagn dcbdiagns dcbdiagn	DCB=DIAGNS=TRACE	DALDIAGN	none		Enable diagnostic trace	
dir	SPACE=(u,(p,s,d))	DALDIR	int		Number of directory blocks	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
disp status stats	DISP=	DALSTATS	char *	Res Word	Data set allocation status (NEW, OLD, etc.)	
dsname dsnam dsn	DSN=	DALDSNAM	char[47]	Dsname	Name of data set to allocate	
dsntype dsnt	DSNTYPE=	DALDSNT	char *	Res Word	Special data set type	
dsorg dcbdsorg	DCB=DSORG=	DALDSORG	char *	Res Word	Data set organization	
dummy	DUMMY	DALDUMMY	none		Allocate dummy data set	
erropt eropt dcberropt dcberopt	DCB=EROPT=	DALEROPT	char *	Res Word	Error handling option	
expiration expires labelexpdt expdt expdl	LABEL=EXPDT=	DALEXPDT DALEXPDL	char *		Expiration date	**(2)**
fcb fcbim	FCB=	DALFCBIM	char[5]		Printer FCB image name	
fcbalign fcbverify fcbav	FCB= (,ALIGN/VERIFY)	DALFCBAV	char *	Res Word	Request FCB alignment or verification	
flashcount fcnt	FLASH=(o,count)	DALFCNT	int		Number of SYSOUT copies to be flashed	
flashforms flashform flash fform	FLASH=	DALFFORM	char[5]		Forms overlay name	
freeclose close	FREE=CLOSE	DALCLOSE	none		Free file when closed	
func dcbfunc	DCB=FUNC=	DALFUNC	char *	Res Word	Card reader/punch function	
gncp dcbgncp	DCB=GNCP=	DALGNCP	int		Number of channel programs for GAM	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
hold shold	HOLD=YES	DALSHOLD	none		Hold SYSOUT for later processing	
interchange inchg	none	DALINCHG	int		Specify volume interchange media type	
interval intvl dcbintvl	DCB=INTVL=	DALINTVL	int		TCAM invitation interval	
ipltxtid ipltxid ipltx dcbipltxtid dcbipltxid dcbipltx	DCB=IPLTXID=	DALIPLTX	char[9]		3705 IPL text id	
keylen keyln kylen dcbkeylen dcbkeyln dcbkylen	DCB=KEYLEN=	DALKYLEN	int		Key length	
keyoff keyo	KEYOFF=	DALKEYO	int		VSAM key offset	
labelinout labelin labelout inout	LABEL=(,,IN/OUT)	DALINOUT	char *	Res Word	Input-only or output-only	
labelpassword labelpswd paspr	LABEL=(,,pw)	DALPASPR	char *	Res Word	Password protection specification	
labelseq fileseq fileno dsseq	LABEL=seq	DALDSSEQ	int		File sequence number	
labeltype label	LABEL=(,lt)	DALLABEL	char *	Res Word	Type of tape label	
like	LIKE=	DALLIKE	char[47]	Dsname	Name of a model data set (SMS)	
limct dcblimct	DCB=LIMCT=	DALLIMCT	int		BDAM search limit	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
lreclk lreck dcblreclk dcblreck	DCB=LRECL=lrK	DALLRECK	none		ANSI tape LRECL is expressed in K	
lrecl dcblrecl	DCB=LRECL=	DALLRECL	int		Logical record length	**(3)**
member membr	DSN=ds(mem)	DALMEMBR	char[9]		PDS member name	
mgmtclass mgmtclas mgcl	MGMTCLAS=	DALMGCL	char[9]		SMS management class	
mode dcbmode	DCB=MODE=	DALMODE	char *	Res Word	Card reader/punch mode	
modify mmod	MODIFY=	DALMMOD	char[4]		3800 printer copy modification module	
modifytrc mtrc	MODIFY=(,trc)	DALMTRC	int		3800 printer table reference character	
ncp dcbncp	DCB=NCP=	DALNCP	int		Number of channel programs	
ndisp	DISP=(s,n)	DALNDISP	char *	Res Word	Normal disposition	
optcd dcboptcd	DCB=OPTCD=	DALOPTCD	char *	Res Word	Optional access method service codes	
outlim outlm	OUTLIM=	DALOUTLM	int		Output limit for SYSOUT file	
output outpt	OUTPUT=	DALOUTPT	char[27]		Name of OUTPUT statement	
parallel unitp paral	UNIT=(,P)	DALPARAL	none		Mount volumes in parallel	
password passw	none	DALPASSW	char[9]		Data set password	
path	PATH=	DALPATH	char[256]		HFS path name	**(4)**
pathcdisp pcdisp pcds cnds	PATHDISP=(n,c)	DALPCDS	char *	Res Word	Conditional (ABEND) disposition for HFS file	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
pathmode pmode pmde	PATHMODE=	DALPMDE	char *	Multiple Res Word	HFS file permissions	
pathndisp pathdisp pndisp pnds	PATHDISP=	DALPNDS	char *	Res Word	Normal disposition for HFS file	
pathopts pathopt popts popt	PATHOPTS=	DALPOPT	char *	Multiple Res Word	HFS file options	
pcir dcbpcir	DCB=PCI=(r,s)	DALPCIR	char *	Res Word	PCI handling for receiving	
pcis dcbpcis	DCB=PCI=(r,s)	DALPCIS	char *	Res Word	PCI handling for sending	
permalloc permanent perma perm	none	DALPERMA	none		Allocate permanently	
private privt	VOL=(PRIVATE,. . .)	DALPRIVT	none		Private volume required	
protect prot	PROTECT=YES	DALPROT	none		Protect data set with RACF	
prtsp dcbprtsp	DCB=PRTSP=	DALPRTSP	char *	Res Word	Printer spacing	
qname	QNAME=	DALQNAME	char[18]		TCAM queue name	
recfm dcbrecfm	DCB=RECFM=	DALRECFM	char *	Res Word	Record format	
recorg reco	RECORG=	DALRECO	char *	Res Word	Organization of VSAM file	
refdd refd	REFDD=	DALREFD	char[27]		Copy DCB attributes from DDname	
reserve1 reserve rsrvf dcbreserve1 dcbreserve dcbrsrvf	DCB=RESERVE= (r1,r2)	DALRSRVF	int		Number of bytes to be reserved in first buffer	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
reserve2 rsrvs dcbreserve2 dcbrsrvs	DCB=RESERVE= (r1,r2)	DALRSRVS	int		Number of bytes to be reserved in later buffers	
retddname rtddname retddn rtddn	none	DALRTDDN	char(*)[9]		Return DDname allocated	
retdsnam rtdsname retdsn rtdsn	return DSN=	DALRTDSN	char(*)[45]		Return dsname allocated	
retdsorg rtdsorg retorg rtorg	return DCB=DSORG=	DALRTORG	int *	Encoded	Return data set organization	
retention labelretp retpd	LABEL=RETPD=	DALRETPD	int		Retention period	
retvolume retvolser retvol rtvolume rtvolser rtvol	Return VOL=SER=	DALRTVOL	char(*)[7]		Return volume serial	
secmodel secm	SECMODEL=	DALSECM	char[47]		SMS security model data set	**(5)**
segment segm	SEGMENT=	DALSEGM	int		SYSOUT segment page count	
spaceformat spform spfrm	SPACE=(,,,form)	DALSPFRM	char *	Res Word	Format of allocated space	
spacerlse release rlse	SPACE=(,,RLSE)	DALRLSE	none		Release unused space	
spaceround round	SPACE=(,,,,ROUND)	DALROUND	none		Round up space request	

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
space1 space primary prime	SPACE=(u,(s1,s2))	DALPRIME	int		Primary space allocation	
space2 secondary extend secnd	SPACE=(u,(s1,s2))	DALSECND	int		Secondary space allocation	
spin	SPIN=	DALSPIN	char *	Res Word	Determine when freed SYSOUT should be printed	
stack dcbstack	DCB=STACK=	DALSTACK	int		Card punch stacker	
storclass storclas stcl	STORCLAS=	DALSTCL	char[9]		SMS storage class	
sysout sysou	SYSOUT=	DALSYSOU	char[2]		Optional Sysout class	
sysoutforms sysoutformno sysoutform formno forms form sfmno	SYSOUT=(c,,f)	DALSFMNO	char[5]		SYSOUT forms number	
sysoutpgmname sysoutpgmnm sysoutpgm sysoutwtr programname pgmname wtrname spgnm	SYSOUT=(c,p)	DALSPGNM	char[9]		SYSOUT writer program name	
subsys ssnm	SUBSYS=	DALSSNM	char[5]		Subsystem name	
subsysattr ssattr ssatt	none	DALSSAT	char *	Res Word	Subsystem attributes	**(6)**
subsysparm ssparm ssprm	SUBSYS=(,parm…)	DALSSPM	char[68]	Multiple	Subsystem parameters	**(4)**

continued

Table 3.2 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
terminal termts term	TERM=TS	DALTERM	none		Allocate TSO terminal	
thresh thrsh dcbthresh dcbthrsh	DCB=THRESH=	DALTHRSH	int		TCAM message queue threshhold	
tracks track trk	SPACE=(TRK,. . .)	DALTRK	none		Allocate space in tracks	
trtch dcbtrtch	DCB=TRTCH	DALTRTCH	char *	Res Word	Tape recording technique	
ucs	UCS=	DALUCS	char[5]		SYSOUT UCS or print train	
ucsfold ufold fold	UCS=(,FOLD)	DALUFOLD	none		Fold to upper case	
ucsverify uverify verify uvrfy	UCS=(,,VERIFY)	DALUVRFY	none		Request UCS verification	
volcount vlcnt	VOL=(,,,count)	DALVLCNT	int		Maximum number of volumes	
volrefdsname volrefdsn vlrds	VOL=REF=	DALVLRDS	char[46]	Dsname	Request same volume as another data set	
volseq vlseq	VOL=(,,seq)	DALVLSEQ	int		Sequence number of first volume to mount	
volume volser vol vlser	VOL=SER=	DALVLSER	char[7]	Multiple	Names of required volumes	

(1) A value of L may be specified for this option in the keywords string. If the value is specified as an extra argument, it must be numeric. An argument value of 0x80 is interpreted as L.

(2) An expiration date may be specified in one of three formats. In a five-digit expiration date, the first two digits are interpreted as the years since 1900. In a seven-digit expiration date, the first four digits are the entire year. The format "yyyy/ddd", as used with the JCL EXPDT keyword, is also supported.

(3) A value of X may be specified for this option in the keywords string. If the value is specified as an extra argument, it must be numeric. An argument value of 0x8000 is interpreted as X.

(4) This parameter value is not translated to upper case.

(5) This keyword has the same syntax as the JCL SECMODEL keyword. That is, it may be specified either as "dsname" or as "(dsname,GENERIC)". In the latter format, the GENERIC must be present in the keyword value and cannot be specified as an extra argument. With either format, if the "?" notation is used, the extra argument may specify only the data set name. That is, the following calls are correct:

```
osdynalloc(DYN_ALLOC, "secmodel=?", NULL, "SYS1.PARMLIB");
osdynalloc(DYN_ALLOC, "secmodel=(?,GENERIC)", NULL, "SYS1.PARMLIB");
```

while the following are not supported:

```
osdynalloc(DYN_ALLOC, "secmodel=?", NULL, "(SYS1.PARMLIB,GENERIC)");
osdynalloc(DYN_ALLOC, "secmodel=(?,?), NULL, "SYS1.PARMLIB", "GENERIC");
```

(6) The only value currently accepted for this keyword is SYSIN (requesting a SYSIN data set).

Table 3.3 *Dynamic Free Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
ddname ddnam ddn	none	DUNDDNAM	char[9]		DDname to free	
dest destnode node ovsus	DEST=	DUNOVSUS	char[9]		SYSOUT destination node	
destuser userid user ovuid	DEST=(n,user)	DUNOVUID	char[9]		SYSOUT destination userid	
disp ndisp ovdsp	DISP=	DUNOVDSP	char *	Res Word	Disposition (KEEP, DELETE, etc)	
dsn dsnam dsname	DSN=	DUNDSNAM	char[47]	Dsname	Name of data set to free	
hold ovshq	HOLD=YES	DUNOVSHQ	none		Hold freed SYSOUT	
member membr	DSN=ds(mem)	DUNMEMBR	char[9]		Member name to free	
nohold ovsnh	HOLD=NO	DUNOVSNH	none		Do not hold freed SYSOUT	
notinuse remove remov	none	DUNREMOV	none		Remove in-use even if permanently allocated	

continued

Table 3.3 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
path	PATH=	DUNPATH	char[256]		HFS path name to free	**(1)**
pathdisp pathndisp ovpds	PATHDISP=	DUNOVPDS	char *	Res Word	Disposition of HFS file	
spin	SPIN=	DUNSPIN	char *	Res Word	Print freed SYSOUT immediately	
sysoutclass sysout ovcls	SYSOUT=	DUNOVCLS	char[2]		Overriding SYSOUT class	
unallocate unalc	none	DUNUNALC	none		Free even if permanently allocated	

(1) The value for this keyword is not translated to upper case.

Table 3.4 *Dynamic Concatenation Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
ddnames ddname ddnam ddn	none	DDCDDNAM	char[9]	Multiple	DDnames to concatenate	
permanent permc perm	none	DDCPERMC	none		Concatenate permanently	

Table 3.5 *Dynamic Deconcatenation Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
ddname ddnam ddn	none	DDCDDNAM	char[9]		DDname to deconcatenate	

Table 3.6 *Dynamic Mark Not-in-use Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
current curnt	none	DRICURNT	none		Remove in-use for all allocations except for the current TCB	
tcbaddr tcbad tcb	none	DRITCBAD	void *		Remove in-use for allocations of this TCB	

Table 3.7 *Dynamic DDname Allocation Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
ddname ddnam ddn	none	DDNDDNAM	char[9]		DDname to be allocated	
retdummy rtdummy retdum rtdum	return DUMMY	DDNRTDUM	int *	Encoded	Return DUMMY status	

Table 3.8 *Dynamic Allocation Inquiry Keywords*

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
ddname ddnam ddn	none	DINDDNAM	char[9]		DDname for which information is needed	
dsname dsnam dsn	DSN=	DINDSNAM	char[47]	Dsname	Data set for which information is needed	
path	PATH=	DINPATH	char[256]		HFS file name for which information is needed	**(1)**
relno	none	DINRELNO	int		Relative allocation number for which information is needed	
retattr rtattr rattr retatt rtatt ratt	none	DINRTATT	int *	Encoded	Return attributes of the allocation	
retavgrec rtavgrec ravgrec retavgr rtavgr ravgr	return AVGREC=	DINRAVGR	int *	Encoded	Return AVGREC specification	
retcdisp rtcdisp rcdisp retcdp rtcdp rcdp	return DISP=(,,c)	DINRTCDP	int *	Encoded	Return conditional disposition	

continued

Table 3.8 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
retcntl rtcntl rcntl	return CNTL=	DINRCNTL	char(*)[27]		Return referenced CNTL statement	
retdataclass rtdataclass rdataclass retdataclas rtdataclas rdataclas retdacl rtdacl rdacl	return DATACLAS=	DINRDACL	char(*)[9]		Return SMS data class	
retddname rtddname rddname retddn rtddn rddn	none	DINRTDDN	char(*)[9]		Return DDname	
retdsname rtdsname rdsname retdsn rtdsn rdsn	none	DINRTDSN	char(*)[45]		Return data set name	
retdsntype rtdsntype rdsntype retdsnt rtdsnt rdsnt	return DSNTYPE=	DINRDSNT	int *	Encoded	Return data set type information	
retdsorg rtdsorg rdsorg retorg rtorg rorg	return DCB=DSORG=	DINRTORG	int *	Encoded	Return data set organization	
retkeyoff rtkeyoff rkeyoff retkeyo rtkeyo rkeyo	return KEYOFF=	DINRKEYO	int *		Return VSAM key offset	

continued

Table 3.8 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
retlast rtlast rlast retlst rtlst rlst	none	DINRTLST	int *	Encoded	Return last allocation indication	
retlike rtlike rlike	return LIKE=	DINRLIKE	char(*)[45]		Return name of model data set	
retlimit rtlimit rlimit retlim rtlim rlim	none	DINRTLIM	int *		Return number of allocations over the limit	
retmember rtmember rmember retmem rtmem rmem	return DSN= ds (mem)	DINRTMEM	char(*)[9]		Return member name	
retmgmtclass rtmgmtclass rmgmtclass retmgmtclas rtmgmtclas rmgmtclas retmgcl rtmgcl rmgcl	return MGMTCLAS=	DINRMGCL	char(*)[9]		Return SMS management class	
retndisp rtndisp rndisp retndp rtndp rndp	return DISP=(,n)	DINRTNDP	int *	Encoded	Return normal disposition	

continued

Table 3.8 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
retpathcdisp rtpathcdisp rpathcdisp retpcdisp rtpcdisp rpcdisp retpcds rtpcds rpcds retcnds rtcnds rcnds	return PATHDISP=(n,c)	DINRCNDS	int *	Encoded	Return HFS file conditional disposition	
retpathmode rtpathmode rpathmode retpmode rtpmode rpmode retpmde rtpmde rpmde	return PATHMODE=	DINRPMDE	int *	Encoded	Return HFS file permissions	
retpathopts rtpathopts rpathopts retpathopt rtpathopt rpathopt retpopts rtpopts rpopts retpopt rtpopt rpopt	return PATHOPTS=	DINRPOPT	int *	Encoded	Return HFS file options	
retrecorg rtrecorg rrecorg retreco rtreco rreco	return RECORG=	DINRRECO	int *	Encoded	Return VSAM file organization	
retrefdd rtrefdd rrefdd retrefd rtrefd rrefd	return REFDD=	DINRREFD	char(*)[27]		Return REFDD DDname	

continued

Table 3.8 (*continued*)

Identifier	JCL Equiv	SVC 99 Key	Value	Format	Description	Notes
retsecmodel rtsecmodel rsecmodel retsecm rtsecm rsecm	return SECMODEL=	DINRSECM	struct *		Return SMS security model information	**(2)**
retsegment rtsegment rsegment retsegm rtsegm rsegm	return SEGMENT=	DINRSEGM	int *		Return SYSOUT segment page count	
retspin rtspin rspin	return SPIN=	DINRSPIN	int *	Encoded	Return allocation SPIN specification	
retstatus rtstatus rstatus retsta rtsta rsta	return DISP=stat	DINRTSTA	int *	Encoded	Return allocation status	
retstorclass rtstorclass rstorclass retstorclas rtstorclas rstorclas retstcl rtstcl rstcl	return STORCLAS=	DINRSTCL	char(*)[9]		Return SMS storage class	
rettype rttype rtype rettyp rttyp rtyp	return DUMMY/*/ SYSOUT/TERM	DINRTTYP	int *	Encoded	Return special allocation type information	

(1) The value for this keyword is not translated to upper case.

(2) The value for the **``retsecmodel''** keyword is a pointer to an object of type **`struct secmodel`**, defined in **`<os.h>`**. This structure includes a field **`profile`**, in which the model profile name will be stored, and a field **`generic`**, in which an encoded indication of whether the profile is generic will be stored. See the IBM publication, the *Authorized Assembler Language Programs* for encoding information.

EXAMPLE

This example uses the inquiry function of SVC 99 to determine the name of the file allocated to the DDname SYSIN. It then derives the name of an object data set from the returned name, and allocates that name to the DDname SYSLIN.

```
#include <os.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main() {
   char dsname[45];
   char member[9];
   char keywords[100];
   char pathname[256];
   int reason = 0;
   int rc;

   rc = osdynalloc(DYN_INQUIRE,
                   "ddn=sysin,retdsn=?,retmem=?,retpath=?,reason=?",
                   NULL, dsname, member, pathname, &reason);
   if (rc < 0) abort();         /* if input parm error */
   if (rc != 0) {
      if (reason == 0x0438)     /* DDname not found */
         printf("SYSIN is not allocated.\n");
      else printf("osdynalloc inquiry failed, rc = %d, "
                  "reason = %04x\n",
                  rc, reason);
      exit(EXIT_FAILURE);
   }

   if (dsname[0] != 0) { /* the file is a data set */
      int l = strlen(dsname);
      char *lastdot;
      if (l > 4 && memcmp(dsname+l-4, ".OBJ", 4) == 0) {
         printf("SYSIN appears to be a .OBJ dataset.\n");
         exit(EXIT_FAILURE);
      }
      lastdot = strrchr(dsname, '.');   /* find final qualifier */
      if (!lastdot) lastdot = dsname+strlen(dsname);
      if (lastdot+4 > &dsname[44]) {
         printf("Object data set name too long.\n");
         exit(EXIT_FAILURE);
      }
      strcpy(lastdot, ".OBJ");          /* replace with .OBJ */
      sprintf(keywords,
              "ddn=sysin,dsn=%s%s%s,disp=shr,reason=?",
              dsname, (member[0]? ",member=": ""), member);
                                 /* build keywords, with or without
                                    a member name */
```

```
      rc = osdynalloc(DYN_ALLOC, keywords, NULL, &reason);
      if (rc < 0) abort();
      if (rc != 0) {
         printf("osdynalloc dsn allocation failed, rc = %d, "
                "reason = %04x\n",
                rc, reason);
         exit(EXIT_FAILURE);
      }
   } else {      /* else, an HFS path name */
      int l = strlen(pathname);
      char *lastdot;
      if (l > 2 && memcmp(pathname+l-2, ".o", 2) == 0) {
         printf("SYSIN appears to be a .o HFS file.\n");
         exit(EXIT_FAILURE);
      }
      lastdot = strrchr(pathname, '.');   /* find extension */
      if (!lastdot) lastdot = pathname+strlen(pathname);
      if (lastdot+2 > &pathname[255]) {
         printf("Object data set name too long.\n");
         exit(EXIT_FAILURE);
      }
      strcpy(lastdot, ".o");            /* replace with .o */
      rc = osdynalloc(DYN_ALLOC,
                      "ddn=syslin,path=?,pathdisp=keep,"
                      "pathopts=ordonly,reason=?", NULL,
                      pathname, &reason);
      if (rc < 0) abort();
      if (rc != 0) {
         printf("osdynalloc path allocation failed, rc = %d, '
                "reason = %04x\n",
                rc, reason);
         exit(EXIT_FAILURE);
      }
   }

   puts("SYSLIN successfully allocated.");
   exit(EXIT_SUCCESS);
}
```

4 MVS Multitasking and Other Low-Level System Interfaces

4-1 Introduction
4-1 Multitasking SAS/C Applications
4-2 Compatibility Changes
4-3 Function Descriptions

Introduction

SAS/C Software provides a number of C macros and functions that allow you to issue common MVS and CMS system macros without having to code any assembler language. Interfaces are provided supporting low-level memory allocation, terminal I/O, and multitasking (MVS only). Most of these macros and functions can be used with either the standard SAS/C Library or with the Systems Programming Environment (SPE). In a few cases, macros can be used only with SPE, due to the existence of conflicting functionality in the standard SAS/C Library. (For instance, the **`STIMER`** macro conflicts with the standard alarm function.) Declarations for the SPE-only functions are collected in the header file **`<spetask.h>`**.

These interfaces provide little functionality beyond that available using standard system assembler macros like ATTACH and DETACH. Their purpose is the convenience of avoiding assembler interface routines rather than functional enhancements. The syntactic form of the SAS/C low-level macros and functions has been defined to be as similar as possible to the assembler macros, which facilitates the use of the assembler documentation. (See IBM's *MVS/ESA Application Development Reference: Services for Assembler Language Programs* and *VM/ESA CMS Application Development Reference for Assembler*.)

Multitasking SAS/C Applications

Note that when you run multiple SAS/C tasks in an address space, each task must have its own C framework. That is, each subtask load module must include a **`main`** function, or must use the **`indep`** compiler option to create a new C framework. Each SAS/C framework in an address space operates completely independently of the others. This means that you can use objects created by the library only under the task by which they were created. Specifically:

- You cannot allocate storage with **`malloc`** or **`palloc`** in one framework and free it with **`free`** or **`pfree`** in another.
- You cannot dynamically load a SAS/C load module in one framework and call it or unload it in another.
- You cannot open a C file (using either UNIX-style or standard I/O) in one framework and use it in another.
- Depending on the application, it may be necessary to redirect standard files for SAS/C subtasks. If the standard files are allocated to the terminal or to MVS SYSOUT, generally no problems will result from using them from all subtasks; but if **`stdout`** or **`stderr`** is a disk file, lost output and or I/O errors may occur.
- Each framework has its own defined run-time options. Run-time options are not automatically inherited, and must be set explicitly (either by the caller or by use of _options in the subtask) if the defaults are not suitable.

- It is possible, in most situations, to run the SAS/C Debugger in several tasks at the same time. However, this is not recommended. A separate invocation of the debugger is needed for each task, which significantly increases total memory requirements. In addition, it is often difficult to control which debugger will receive input commands. (For this reason, running the debugger in line mode in this situation is recommended.)
- Only external and permanent scope environment variables are shared between frameworks.
- OpenEdition resources such as file descriptors and signal handlers are shared between all tasks in an address space. For this reason, it is recommended that no more than one task in an address space use OpenEdition functionality.

Note: Many of these restrictions can be bypassed by using operating system facilities directly. For instance, memory allocated by GETMAIN and DCBs processed using BSAM I/O can be shared freely among subtasks so long as all MVS restrictions are honored.

Compatibility Changes

With Release 5.50, some of the symbols defined in the **<tput.h>** and **<bldexit.h>** header files have been changed. To avoid conflicts with **<ostask.h>** or **<spetask.h>**, you must replace some of the symbols that were defined in prior releases of the SAS/C Compiler.

The changes shown in Table 4.1 on page 4-2 apply to the **TPUT** and **TGET** macros defined in **<tput.h>**.

Table 4.1
<tput.h> Changes

Old Form	New Form
EDIT	_EDIT
ASIS	_ASIS
CONTROL	_CONTROL
FULLSCR	_FULLSCR
WAIT	_WAIT
NOWAIT	_NOWAIT
HOLD	_HOLD
NOHOLD	_NOHOLD
BREAKIN	_BREAKIN
NOBREAK	_NOBREAK
HIGHP	_HIGHP
LOWP	_LOWP

The changes shown in Table 4.2 apply to the **bldexit** function defined in **<bldexit.h>**.

Table 4.2
<bldexit.h> Changes

Old Form	New Form
ASYNCH	_ASYNCH
NOR13	_NOR13
FLOATSV	_FLOATSV
AMODE	_AMODE

Function Descriptions

Descriptions of the low-level multitasking functions follow. Each description includes a synopsis, description, discussion of return values and portability issues, and an example. Also errors, cautions, diagnostics, implementation details, and usage notes are included where appropriate.

ABEND Abnormally Terminate a Program

SYNOPSIS

```
#include <spetask.h>

void ABEND(unsigned code, unsigned reason, dumpopt, sysopt);
```

DESCRIPTION

The SAS/C **ABEND** macro implements the functionality of the MVS Assembler ABEND macro. The **code** argument specifies the ABEND code, and the **reason** argument specifies the reason code. **dumpopt** may be specified as either DUMP or NODUMP, to indicate whether a dump should be produced. **sysopt** may be specified as either USER or SYSTEM to indicate whether the ABEND is a system ABEND or a user ABEND.

CAUTIONS

ABEND is intended for use only in the SPE environment. The library functions **abort** or **abend** should be used to force abnormal termination when using the full library.

EXAMPLE

This example terminates the application with system ABEND code **BAD** and reason code **0**.

```
#include <spetask.h>

ABEND(0xbad,0,DUMP,SYSTEM);
```

RELATED FUNCTIONS

abend, **abort**

ATTACH Create a New Subtask

SYNOPSIS

```
#include <ostask.h>

int ATTACH(void **tcbaddr, ...);
```

DESCRIPTION

The **ATTACH** function implements the functionality of the MVS assembler ATTACH macro. The **tcbaddr** argument is the address of a **void *** area where the address of the new task control block (TCB) is to be stored. The remainder of the argument list is a list of keywords followed, in most cases, by an argument specifying a value for the keyword. The list is terminated by the **_Aend** keyword.

The supported keywords and their associated data are as follows:

- The **_Aep** keyword is equivalent to the Assembler EP keyword. The next argument should be a null-terminated string containing the name of the load module to attach. The name is translated to uppercase and padded to eight characters.
- The **_Adcb** keyword is equivalent to the Assembler DCB keyword. The next argument should be a pointer to an open DCB. (A DSORG=PO DCB obtained using the **osdcb** and **osbopen** functions may be used.)
- The **_Alpmod** keyword is equivalent to the Assembler LPMOD keyword. The next argument should be an **unsigned char** value to be subtracted from the limit priority of the new task.
- The **_Adpmod** keyword is equivalent to the Assembler DPMOD keyword. The next argument should be a signed integer value to be added to the dispatching priority of the new task.
- The **_Aparam** keyword is equivalent to the Assembler PARAM keyword. The next argument should be a pointer to a list of arguments to be passed to the new task. (This value is loaded into register 1 before the new task is attached.) If no parameter list is specified, a value of **0** is placed in register 1 before the **ATTACH** function is called. Many programs do not tolerate a 0 parameter pointer, and if you are not certain of the parameter list requirements of the program you are attaching, a parameter list containing a pointer to a halfword of zeroes is recommended. If you are attaching a SAS/C program, you should not pass a 0 parameter pointer. (See "Calling a C Program from Assembler" in Chapter 11 of the *SAS/C Compiler and Library User's Guide, Third Edition*, for more information on passing parameters to SAS/C load modules.)
- The **_Aecb** keyword is equivalent to the Assembler ECB keyword. The next argument should be a pointer to a fullword, which is an ECB to be posted by the POST macro when the subtask terminates.
- The **_Aetxr** keyword is equivalent to the Assembler ETXR keyword. The next argument should be a **__remote** function pointer that specifies the address of a C function to be called when the subtask terminates. Linkage for this function is as follows:

```
void etxr(void *tcbaddr);
```

ATTACH Create a New Subtask
(continued)

The function argument is the address of the terminated TCB. The ETXR routine receives control in SPE as a **bldexit** routine. In the full library implementation the ETXR routine can receive control anytime an asynchronous signal handler could receive control.

- The **_Ashspl** keyword is equivalent to the Assembler SHSPL keyword. The next argument should be a **char ***, pointing to a list of subpool numbers preceded by the number of subpools in the list. Subpool 78 is always shared with subtasks created by the SAS/C **ATTACH** function, whether explicitly specified or not.
- The **_Aszerono** keyword is equivalent to the Assembler keyword SZERO=NO. This keyword does not take a value. The next argument should be another keyword.
- The **_Atasklib** keyword is equivalent to the Assembler TASKLIB keyword. The next argument should be a pointer to an open DCB. (A DSORG=PO DCB obtained using the **osdcb** and **osbopen** functions may be used.)
- The **_Aend** keyword indicates the end of the list of keywords.

The following should be noted about subtask termination. If neither an **_Aecb** nor **_Aetxr** keyword was specified, the subtask TCB is automatically detached when it terminates. Otherwise, the program is required to detach the TCB itself after the subtask has terminated. (Refer to the **DETACH** macro later in this chapter.)

If a C program terminates normally with active subtasks created by the **ATTACH** function, the library detaches these subtasks automatically. If a subtask has terminated, the library assumes that the program has detached it. If this assumption is untrue, the program will be abnormally terminated by the operating system.

If an active subtask is detached by the program using the C **DETACH** macro, an ETXR routine is not called. Also, ETXR routines are not called for tasks detached by the library during normal program termination.

RETURN VALUE

ATTACH returns **0** if the ATTACH macro was successful. If the ATTACH macro fails, it returns the return code from the macro, which will be a positive value. **ATTACH** may also return **-1** to indicate an unknown keyword, or **-2** if there was not enough memory to do the attach.

CAUTIONS

In general, the **_Aparam** keyword should always be specified. See the discussion of this keyword under "DESCRIPTION."

You should be wary of defining subtask parameters in automatic storage, unless the calling function waits for the subtask to complete. Otherwise, the storage for the parameters could be freed or reused while the subtask is accessing them.

IMPLEMENTATION

The **ATTACH** function is implemented by the source module L$UATTA. The SPE implementation of ETXR exits is provided by the source module L$UAEOT.

ATTACH Create a New Subtask
(continued)

EXAMPLE

This example attaches the SAS/C Compiler out of the DDname SASCLIB, and uses the **WAIT1** function to wait for it to complete.

```
#include <ostask.h>
#include <stdio.h>
#include <osio.h>

DCB_t *tasklib;
unsigned myecb;
int rc;
void *subtcb;
void *plist [1];                    /* parm list for compiler     */
struct {
   short len;
   char options[8];
} argument = {8, "OPTIMIZE"};

tasklib = osdcb("SASCLIB", "DSORG=PO", NULL, 0);
if (!tasklib) abort();
if (osbopen(tasklib, "input")) abort();
myecb = 0;                          /* Initialize ECB.            */
plist [0] = (void *)(0x80000000 | (unsigned) &argument);
   /* Set up compiler parm list.                                    */
rc = ATTACH(&subtcb, _Aep, "LC370B", _Atasklib, tasklib, _Aecb,
            &myecb, _Aparam, plist, _Aend);
if (rc != 0) abort();            /* if the ATTACH failed          */
WAIT1(&myecb);                   /* Wait for compiler to complete.*/
DETACH(&subtcb,NOSTAE);          /* Detach the TCB.               */
printf("Compiler return code %d\n",
        /* Display compiler return code.                          */
        myecb & 0x3fffffff);
osbclose(tasklib, "disp", 1);
```

RELATED FUNCTIONS

DETACH, **oslink**, **system**

CHAP Change a Task's Priority

SYNOPSIS

```
#include <ostask.h>

void CHAP(int prio, void **tcbaddr);
```

DESCRIPTION

The SAS/C **CHAP** macro implements the functionality of the MVS Assembler CHAP macro. The **prio** argument is the amount by which the priority of the specified task should be changed. The **tcbaddr** argument is the address of a fullword containing the address of the TCB whose priority is to be changed or `0` if the priority of the active task is to be changed.

CMSSTOR Allocate or Free Storage

SYNOPSIS

```
#include <cmsstor.h>

int CMSSTOR_OBT(unsigned int bytes, void **loc, char *subpool,
                char options, int erresp);

int CMSSTOR_REL(unsigned int bytes, void *loc2, char *subpool,
                char options, int erresp);
```

DESCRIPTION

These functions simulate the CMSSTOR macro. **CMSSTOR_OBT** requests a fixed amount of free storage. **CMSSTOR_REL** releases free storage that has been allocated by **CMSSTOR_OBT**.

Each macro causes the compiler to generate an SVC 204, with register 1 addressing an appropriate parameter list. Most of the arguments correspond directly to the equivalent assembler macro parameters.

bytes
is the number of bytes of free storage to be allocated by **CMSSTOR_OBT** or released by **CMSSTOR_REL**.

loc
is a pointer to a **void *** where the address of the first byte of allocated storage can be stored.

loc2
is a pointer to the storage to be freed.

subpool
is a pointer to a null-terminated string containing the name of the CMS subpool from which storage is to be allocated or freed. If the subpool is specified as a null string, storage is allocated from the subpool **''CMS''** or released from the subpool in which it was allocated. The subpool must be a PRIVATE or SHARED subpool, not a GLOBAL subpool.

options
is a set of option flags. Option flags are defined as macros in **<cmsstor.h>**. There are three types of option flags, corresponding to the assembler macro keyword parameters MSG, LOC, and BNDRY.

MSG options	indicate whether or not a message is to be issued if a failure occurs. There are two MSG flags: **MSG_YES** and **MSG_NO**.
LOC options	specify the location of the storage requested. There are four LOC flags: **LOC_ABOVE** specifies storage above the 16-megabyte line, **LOC_BELOW** specifies storage below the 16-megabyte line, **LOC_SAME** specifies storage located according to the addressing mode of the calling program, and **LOC_ANY** specifies any location.
BNDRY options	specify the required alignment for the requested storage. There are two BNDRY flags: **BNDRY_DWORD** requests doubleword alignment, and **BNDRY_PAGE** requests page alignment.

CMSSTOR Allocate or Free Storage

(continued)

Also, the macros **OBT_DEF** and **REL_DEF** are defined. These macros represent the defaults used by the CMSSTOR assembler macro when the MSG, LOC, and BNDRY parameters are omitted. They are defined as

```
MSG_YES+LOC_SAME+BNDRY_DWORD
```

erresp
indicates how a failure is to be handled. Two macros are defined in **<cmsstor.h>**. Use the macro **ERR_RET** to indicate that a nonzero value is to be returned in the event of a failure. Use the macro **ERR_ABN** to indicate that the system should abend in the event of a failure.

RETURN VALUE

Each macro returns the value in register 15 after the SVC 204 has completed. For **CMSSTOR_OBT**, the value in register 1 is stored in the **void *** pointed to by **loc**.

ERRORS AND DIAGNOSTICS

The possible errors are the same as when the CMSSTOR Assembler macro is used in an assembler language program.

CAUTION

The macros do not perform any error checking on their arguments. Incorrect arguments such as invalid combinations of option flags are either ignored entirely or remain undetected until execution time, at which point they cause unpredictable and probably undesirable results.

PORTABILITY

None of the macros are portable.

SEE ALSO

See *VM/ESA CMS Application Development Reference for Assembler.*

DMSFREE Allocate or Free DMSFREE Storage

SYNOPSIS

```
#include <dmsfree.h>

int DMSFREE(unsigned int dwords, void **loc, char options,
            int erresp);

int DMSFREE_V(unsigned int dwords, unsigned int min,
              void **loc, unsigned int *got; char options,
              int erresp);

int DMSFRET(unsigned int dwords, void *loc2, char msg,
            int erresp);
```

DESCRIPTION

These functions simulate the CMS DMSFREE and DMSFRET Assembler macros. **DMSFREE** requests a fixed amount of free storage. **DMSFREE_V** requests a variable amount of free storage. **DMSFRET** releases free storage that has been allocated by **DMSFREE** or **DMSFREE_V**.

Each macro causes the compiler to generate an SVC 203, followed by the halfword of flags specified by the macro arguments. Most of the arguments correspond directly to the equivalent assembler macro parameters.

dwords
is the number of doublewords of free storage to be allocated by **DMSFREE** or **DMSFREE_V** or released by **DMSFRET**.

min
is the minimum number of doublewords of free storage to be allocated in a variable request.

loc
is a pointer to a **void *** where the address of the first byte of allocated free storage can be stored.

loc2
is a pointer to the storage to be freed.

got
is a pointer to an **unsigned int** where the number of doublewords actually allocated by a variable request can be stored.

options
is a set of option flags. Option flags are defined as macros in **<dmsfree.h>**. There are three types of option flags, corresponding to the assembler macro keyword parameters TYPE, AREA, and MSG.

TYPE options	specify the type of storage requested. There are two TYPE flags: **TYPE_NUC** specifies NUCLEUS free storage, and **TYPE_USER** specifies USER free storage.
AREA options	specify the area of storage from which the request will be filled. There are three AREA flags: **AREA_LOW** specifies that the request be filled from free storage below the user areas, **AREA_HIGH** specifies that the request be filled from free storage above the user area, and **AREA_ANY** specifies that any area can be used.

DMSFREE Allocate or Free DMSFREE Storage

(continued)

MSG options — indicate whether or not a message is to be issued if a failure occurs. There are two MSG flags: **MSG_YES** and **MSG_NO**.

Also, the macro **FREE_DEF** is defined. This macro represents the defaults used by the DMSFREE assembler macro when the TYPE, AREA, and MSG parameters are omitted. **FREE_DEF** is defined as follows:

```
AREA_ANY+MSG_YES
```

Use the **FREE_DEF** macro as the **options** argument, or create it by using OR to select one choice from each of the AREA, MSG, and (optionally) TYPE flags. It is not possible to allow the AREA or MSG flag to default.

msg
is one of the two MSG flags.

erresp
indicates how a failure is to be handled. Two macros are defined in **<dmsfree.h>**. Use the macro **ERR_RET** to indicate that a nonzero value is to be returned in the event of a failure. Use the macro **ERR_ABN** to indicate that the system should abend in the event of a failure.

RETURN VALUE

Each macro returns the value in register 15 after the SVC 203 has completed. For **DMSFREE** and **DMSFREE_V**, the value in register 1 is stored in the **void *** pointed to by **loc**. For **DMSFREE_V**, the value in register 0 is stored in the **unsigned int** pointed to by **got**.

The possible errors are the same as when the DMSFREE or DMSFRET assembler macros are used in an assembly language program.

CAUTION

The macros do not perform any error checking on their arguments. Incorrect arguments such as invalid combinations of option flags are either ignored entirely or remain undetected until execution time, at which point they cause unpredictable and probably undesirable results.

PORTABILITY

None of the macros are portable.

SEE ALSO

See *VM/SP CMS for System Programming*.

DETACH Remove a Subtask from the System

SYNOPSIS

```
#include <ostask.h>

int DETACH(void **tcbaddr, option);
```

DESCRIPTION

The SAS/C **DETACH** macro implements the functionality of the MVS Assembler DETACH macro. The **tcbaddr** argument is the address of a **void *** area containing the address of the TCB to be detached. The **option** argument must be specified as either STAE or NOSTAE, indicating whether the assembler STAE=YES or STAE=NO option is required.

DETACH must be used for a terminated subtask to remove the TCB from the system. If any subtasks created by the SAS/C **ATTACH** function are still running when the program terminates, these subtasks are detached automatically.

RETURN VALUE

The **DETACH** macro returns the value returned in register 15 by the assembler DETACH macro.

EXAMPLE

See the example for the **ATTACH** macro earlier in this chapter.

RELATED FUNCTIONS

ATTACH

DEQ Release an Enqueued Resource

SYNOPSIS

```
#include <ostask.h>

int DEQ(char *qname, char *rname, int rlen, scope, ret);
```

DESCRIPTION

The SAS/C **DEQ** macro implements the functionality of the MVS Assembler DEQ macro. The **qname** argument specifies the queue name and must be eight or more characters in length; only the first eight characters are used. The **rname** argument specifies the resource name. Both **qname** and **rname** are passed to the DEQ SVC as is. The library does not pad or translate the strings in any way.

The **rlen** argument specifies the length of the **rname**. The **scope** argument specifies whether the scope of the name is the job step, the current system, or all systems in a complex. **scope** must be specified as one of the keywords STEP, SYSTEM, or SYSTEMS. The **ret** argument specifies the same information as the assembler RET keyword and must be either NONE or HAVE.

RETURN VALUE

The SAS/C **DEQ** macro returns the same value as the return code from the assembler DEQ macro.

EXAMPLE

See the example for **ENQ**.

RELATED FUNCTIONS

ENQ

ENQ Wait for a Resource to Become Available

SYNOPSIS

```
#include <ostask.h>

int ENQ(char *qname, char *rname, excl, int rlen, scope, ret);
```

DESCRIPTION

The SAS/C **ENQ** macro implements the functionality of the MVS Assembler ENQ macro. The **qname** argument specifies the queue name and resource name and must be eight or more characters in length; only the first eight characters are used. The **rname** argument specifies the resource name. Both the **qname** and **rname** are passed to the ENQ SVC as is; that is, the library does not pad or translate the strings in any way.

The **excl** argument specifies whether this is an exclusive or shared request and must be specified as either E or S. The **rlen** argument specifies the length of **rname**. The **scope** argument specifies whether the scope of the name is the job step, the current system, or all systems in a complex. It must be specified as one of the keywords STEP, SYSTEM, or SYSTEMS. The **ret** argument specifies the same information as the assembler RET keyword and must be NONE, HAVE, CHNG, USE or TEST.

RETURN VALUE

The SAS/C **ENQ** macro returns the same value as the return code from the assembler ENQ macro.

EXAMPLE

Use the **ENQ** function to obtain shared control of the resource with **qname** MYQN and **rname** FRED.ACCTS.DATA. RET=HAVE is used to prevent ABEND if the resource is unavailable.

```
static char rname [] = "FRED.ACCTS.DATA";
int rc;

rc = ENQ("MYQN    ", rname, S, sizeof(rname)-1, SYSTEM, HAVE);
if (rc != 0)
   printf("ENQ failed!\n");
else {
   process();     /* utilize the resource */
   DEQ("MYQN    ", rname, sizeof(rname)-1, SYSTEM, NONE);
}
```

RELATED FUNCTIONS

DEQ

ESTAE Define an Abnormal Termination Exit

SYNOPSIS

```
#include <spetask.h>

int ESTAE(void *exit, create, void *param, asynch, term);

int ESTAE_CANCEL(void);
```

DESCRIPTION

The SAS/C **ESTAE** and **ESTAE_CANCEL** macros implement the functionality of the MVS assembler ESTAE macro. This macro is supported only with the Systems Programming Environment (SPE).

The **exit** argument of **ESTAE** is the address of code to receive control if the program ABENDs. This argument should ordinarily be specified as a call to the **bldexit** function, whose first argument is the address of the C function that is to be defined as the exit.

create must be either **CT** or **OV**. Specify **CT** if a new ESTAE exit is to be defined, or **OV** if the previous exit is to be overlaid.

param specifies a pointer that is to be made available to the ESTAE exit when it is called.

asynch must be specified as **ASYNCH** or **NOASYNCH**, to indicate whether or not asynchronous interrupts are allowed in the exit.

term must be specified as **TERM** or **NOTERM**. Indicates whether the exit is to be given control for no-retry terminations such as operator cancel.

The **ESTAE_CANCEL** macro has the same effect as the assembler ESTAE macro with an exit address of **0**. That is, it cancels the most recently defined ESTAE exit.

RETURN VALUE

The **ESTAE** and **ESTAE_CANCEL** macros return the ESTAE assembler macro return code in register 15.

CAUTIONS

Use of the **ESTAE** macro in code that uses the full run-time library interferes with library ABEND handling. Additionally, the **bldexit** function cannot be used with the full run-time library.

Similar functionality to **ESTAE** can be obtained with the full library by defining signal handlers for SIGABRT or SIGABND.

ESTAE Define an Abnormal Termination Exit
(continued)

EXAMPLE

This example sets up an ESTAE exit and uses **SETRP** macros within the exit to perform a retry.

```
#include <spetask.h>
#include <bldexit.h>
#include <setjmp.h>

jmp_buf retrybuf;
int rc;

static void estaeex();
extern void msgwtr();            /* message-writing subroutine   */

if (setjmp(retrybuf)) goto retrying;
    /* Set up jump buffer for retry.                             */

rc = ESTAE(bldexit(&estaeex, ASYNCH+NOR13), CT, 0, NOASYNCH, NOTERM);
if (rc != 0) {
   msgwtr("ESTAE macro failed!");
   ABEND(1066, rc, DUMP, USER);       /* Give up after failure.   */
}
.
.
.
retrying:                           /* Retry code here.         */
.
.
.

static void estaeex(void **sa, char **poi) {
   void *SDWA;

   if ((int) sa[2]  == 12)          /* Check R0 for 12.         */
      return;                       /* Give up if no memory.    */
   SDWA = sa[3];                    /* Find the SDWA.           */
   SETRP_DUMP(SDWA, YES);           /* Take dump before retry.  */
      /* request a retry at label retrying above               */
   SETRP_RETRY(bldretry(retrybuf, 1), NOFRESDWA);
   return;
}
```

RELATED FUNCTIONS

bldexit, **SETRP**, **signal**

GETMAIN/FREEMAIN Allocate or Free Virtual Storage

SYNOPSIS

```
#include <getmain.h>

int GETMAIN_C(unsigned int length, int subpool, int options,
              void **loc);

int GETMAIN_U(unsigned int length, int subpool, int options);

int GETMAIN_V(unsigned int max, unsigned int min, int subpool,
              int options, void **loc, unsigned int *alloc);

int FREEMAIN( void **loc, unsigned int length, int subpool,
              int options);
```

DESCRIPTION

These functions simulate the use of the MVS GETMAIN and FREEMAIN Assembler macros. **GETMAIN_C** and **GETMAIN_U** are used to generate conditional and unconditional requests, respectively, to allocate virtual storage. **GETMAIN_V** generates a variable request. **FREEMAIN** generates a request to free virtual storage that has been allocated by one of the **GETMAIN_** macros.

Each macro causes the compiler to generate code to load the appropriate values into registers 0, 1, and 15, and then to generate SVC 120 (under MVS) or SVC 10 (under CMS). The use of each argument is explained here.

length
is the number of bytes of virtual storage to be allocated by the **GETMAIN_** macros or released by **FREEMAIN**.

subpool
specifies the number of the subpool from which the storage is to be allocated.

options
is a set of option flags. All option flags are defined as macros in **<getmain.h>**.

For **GETMAIN_U** and **GETMAIN_C**, *options* is the logical sum of zero or more of the values defined by the macros **BNDRY_PAGE**, **LOC_BELOW**, **LOC_ANY**, and **LOC_RES**. These options are equivalent to the BNDRY and LOC keyword parameters to the GETMAIN assembler macro. For **GETMAIN_V**, *options* also can include COND, indicating a conditional request, or UNCOND, indicating an unconditional request. For **FREEMAIN**, *options* can be either COND or UNCOND.

loc
is a pointer to a **void *** where the address of the allocated storage area can be stored.

max,min
is the maximum and minimum length of the variable request.

alloc
is a pointer to an **unsigned int** where the length allocated for a variable request can be stored.

GETMAIN/FREEMAIN Allocate or Free Virtual Storage

(continued)

RETURN VALUE

GETMAIN_U returns the address of the allocated storage. **GETMAIN_V**, **GETMAIN_C**, and **FREEMAIN** return the value in register 15 after the SVC completes. In addition, **GETMAIN_C** and **GETMAIN_V** store the value in register 1 (the address of the allocated storage) in the **void *** pointed to by **loc**. **GETMAIN_V** also stores the value in register 0 (the amount of virtual storage allocated) in the **unsigned int** addressed by **alloc**.

ERRORS AND DIAGNOSTICS

The possible errors are the same as if the GETMAIN or FREEMAIN assembler macros are used in an assembly language program.

CAUTIONS

The macros do not perform any error checking on their arguments. Incorrect arguments, such as invalid combinations of option flags, are either ignored entirely or remain undetected until execution time, at which point they cause unpredictable and probably undesirable results.

The compiler generates different code sequences depending on the host operating system. If the program is to be executed on a system other than that under which it is compiled, **#define** (or **#undef**) can be used for the symbol **CMS** name before including **<getmain.h>**.

PORTABILITY

None of the macros are portable.

IMPLEMENTATION

If the macro name **CMS** is defined, then CMS versions of the **GETMAIN_U** and **FREEMAIN** macros are defined. These versions generate an inline SVC 10. All option flags are ignored. The **GETMAIN_C** and **GETMAIN_V** macros are not defined.

If the macro name **CMS** is not defined, all four macros are defined. The macros generate an inline SVC 120.

SEE ALSO

See *MVS/ESA Application Development Reference: Services for Assembler Language Programs.*

EXAMPLE

```
   /* Allocate a number of pages of 4K-aligned storage. */
void *pgalloc(int pages, int sp)
{
   return GETMAIN_U((pages*4096),sp,BNDRY_PAGE+LOC_ANY);
}
```

POST Post an ECB

SYNOPSIS

```
#include <ostask.h>

void POST(unsigned *ecbaddr, unsigned code);
```

DESCRIPTION

The SAS/C **POST** macro implements the functionality of the MVS assembler POST macro. The **ecbaddr** argument is the address of the ECB to be POSTed. The **code** argument is the value to be stored in the ECB. The two high-order bits of **code** are ignored.

RELATED FUNCTIONS

WAIT

RDTERM Simulate the RDTERM Assembler Language Macro

SYNOPSIS

```
#include <wrterm.h>

int RDTERM(char *inbuf, short inbufsz, int *inlen);
```

DESCRIPTION

RDTERM simulates the RDTERM assembler language macro. The **RDTERM** macro invokes the **waitrd** macro to produce a parameter list that is equivalent to using the RDTERM assembly language macro with all parameters allowed to default. If the symbol **BIMODAL** is defined, **RDTERM** generates an equivalent to the CMS **LINERD** macro and, therefore, can be used in 31-bit addressing mode.

RETURN VALUE

RDTERM returns the value in register 15 when the **WAITRD** or **LINERD** function returns.

ERRORS AND DIAGNOSTICS

The possible errors are the same as when the RDTERM or LINERD macro is used in an assembler language program.

PORTABILITY

RDTERM is not portable.

IMPLEMENTATION

The definition of **RDTERM** if **BIMODAL** is not defined is as follows:

```
#define RDTERM(b,l,rl) waitrd(b,l,STACK_MIXED,PROMPT_NO,0,0,rl)
```

SEE ALSO

See *VM/SP CMS Command Reference.*

EXAMPLE

```
char *line[130]
int len, rc;
   /* Read up to 130 characters with no prompt and no editing. */
rc = RDTERM(line,sizeof(line),&len);
```

SETRP Set Recovery Parameters in an ESTAE Exit

SYNOPSIS

```
#include <spetask.h>

void SETRP_DUMP(void *sdwa, dumpopt);

void SETRP_COMPCOD(void *sdwa, unsigned code, sysopt);

void SETRP_REASON(void *sdwa, unsigned reason);

void SETRP_RETRY(void *sdwa, void *retry, fresdwa);
```

DESCRIPTION

The SAS/C **SETRP** macros implement the most frequently used functions of the MVS Assembler SETRP macro. These macros should be used only in an ESTAE exit. Each macro's first argument is a pointer to the SDWA (System Diagnostic Work Area), which is passed to the exit by the operating system.

SETRP_DUMP overrides the DUMP specification of the original ABEND. **dumpopt** must be specified as **YES** or **NO** to indicate whether or not a dump should be taken.

SETRP_COMPCOD overrides the completion code of the original ABEND. The **code** argument is the new completion code. The **sysopt** argument must be specified as either **USER** or **SYSTEM**.

SETRP_REASON overrides the reason code of the original ABEND. The **reason** argument is the new reason code.

SETRP_RETRY specifies the address of an ESTAE retry. The **retry** argument is the address at which execution of the retry routine is to begin. To retry the ABEND in C code, the **retry** argument should be a call to the **bldretry** function, whose first argument is a jump buffer defining the retry point. The **fresdwa** argument must be FRESDWA or NOFRESDWA and must indicate whether the SDWA should be freed before control is passed to the retry routine.

CAUTION

The **SETRP** macros should be used only with the Systems Programming Environment (SPE).

EXAMPLE

See the example for **ESTAE**.

RELATED FUNCTIONS

ESTAE, **bldretry**

STATUS Halt or Resume a Subtask

SYNOPSIS

```
#include <ostask.h>

void STATUS(action, void **tcbaddr);
```

DESCRIPTION

The SAS/C **STATUS** macro implements the functionality of the MVS Assembler STATUS macro. The **action** argument specifies whether the task is to be stopped or started and must be either **START** or **STOP**. The **tcbaddr** argument is a pointer to a fullword containing the address of the TCB to be started or stopped.

RELATED FUNCTIONS

ATTACH

STIMER Define a Timer Interval

SYNOPSIS

```
#include <spetask.h>

void STIMER(type, void *exit, unit, void *intvl);
```

DESCRIPTION

The SAS/C **STIMER** macro implements the functionality of the MVS Assembler STIMER macro. This macro is supported only with the Systems Programming Environment (SPE).

type specifies the type of timer interval and must be either **TASK**, **WAIT** or **REAL**.

exit specifies the address of code to be called as a timer exit when the interval is complete. An exit address of **0** means that no exit is to be called. The **exit** argument should ordinarily be specified as a call to the **bldexit** function, whose first argument is the address of the C function which is to be defined as the exit.

unit specifies the units of the timer interval and must be either **TUINTVL**, **BINTVL**, **MICVL**, **DINTVL**, **GMT** or **TOD**.

intvl is a pointer to an area specifying the time interval or time of expiration. The size and interpretation of this data depends on the unit, as described in *Application Development Reference: Services for Assembler Language Programs* available from IBM.

CAUTIONS

Use of the SAS/C **STIMER** macro in code that uses the full run-time library will interfere with library timer handling. Additionally, the **bldexit** function cannot be used with the full run-time library.

Similar functionality to **STIMER** can be obtained with the full library using the functions **alarm**, **sleep**, or **clock**.

RELATED FUNCTIONS

TTIMER, **STIMERM**, **alarm**, **clock**, **sleep**

STIMERM... Define, Test, or Cancel a Real Time Interval

SYNOPSIS

```
#include <spetask.h>

int STIMERM_SET(void *exit, void *parm, unsigned *idaddr, unit,
                void *intvl, wait);

int STIMERM_TEST(unsigned *idaddr, unit, void *intvl);

int STIMERM_CANCEL(unsigned *idaddr, unit, void *intvl);
```

DESCRIPTION

The SAS/C **STIMERM** macros implement the functionality of the MVS Assembler STIMERM macro. These macros are supported only with the Systems Programming Environment (SPE).

STIMER_SET
specifies a real-time interval and how its expiration is to be handled. The **exit** argument is the address of code to be called as a timer exit when the interval is complete. The **exit** argument should ordinarily be specified as a call to the **bldexit** function, whose first argument is the address of the C function which is to be defined as the exit. The **parm** argument is a pointer to be made available to the exit routine when it runs. The **idaddr** argument is the address of an **unsigned** area where an id value can be stored by the STIMERM SVC. The **unit** argument must **TUINTVL**, **BINTVL**, **MICVL**, **DINTVL**, **GMT**, or **TOD**. The **intvl** argument should be a pointer to an area specifying the time interval or time of expiration. The size and interpretation of this data depends on the unit, as described in the IBM publication *Services for Assembler Language Programs.* The **wait** argument indicates whether the program should be suspended until the interval expires and must be either **WAIT** or **NOWAIT**. If you specify **WAIT**, the **exit** argument should be **0**.

STIMER_TEST
tests the amount of time remaining of a real-time interval defined by **STIMERM_SET**. The **idaddr** argument specifies the address of an **unsigned** area containing the ID of the interval to be tested. The **unit** argument indicates the format in which the remaining time is to be stored and must be either **TU** or **MIC**. The **intvl** argument specifies the address where the amount of time remaining should be stored. The size and interpretation of this data depends on the unit, as described in the IBM publication *Application Development Reference: Services for Assembler Language Programs.*

STIMERM_CANCEL
cancels one or more time intervals established with **STIMERM_SET**. The **idaddr** argument specifies the address of an area containing the ID of the timer interval to be cancelled. **idaddr** may also be specified as **0** to cancel all intervals. The **unit** argument specifies the form in which the remaining time is to be stored and must be either **TU** or **MIC**. The **intvl** address specifies the address where the amount of time remaining in the cancelled interval should be stored. It should be specified as **0** if **idaddr** is also **0**. The size and interpretation of the **intvl** data depends on the unit, as described

STIMERM... *Define, Test, or Cancel a Real Time Interval*

(continued)

in the IBM publication *Application Development Reference: Services for Assembler Language.*

RETURN VALUE

The **STIMERM** macros return the value that is returned in register 15 by the assembler STIMERM macro.

CAUTIONS

The **STIMERM** macros should not be used in code that runs with the full library since the **bldexit** function cannot be used in such programs.

Similar functionality to **STIMERM** can be obtained with the full library using the functions **alarm** or **sleep**.

RELATED FUNCTIONS

STIMER, **TTIMER**, **alarm**, **sleep**

TPUT/TGET Terminal I/O via SVC 93

SYNOPSIS

```
#include <tput.h>

int TPUT(void *buffer, unsigned short bufsiz,
         unsigned int options);
int TPUT_ASID(void *buffer, unsigned short bufsiz,
              unsigned short asid, unsigned int options);
int TPUT_USERID(void *buffer, unsigned short bufsiz,
                char *userid, unsigned int options);
int TGET(void *buffer, unsigned short bufsiz,
         unsigned int options);
```

DESCRIPTION

These functions simulate the MVS TPUT and TGET Assembler macros. **TPUT** sends a line of output to the terminal. **TPUT_ASID** and **TPUT_USERID** can be used for interuser communication. **TGET** reads a line of input from the terminal.

Each macro causes the compiler to generate code to load the appropriate values into registers 0, 1, and 15, and then to generate SVC 93. The use of each argument is explained below:

buffer
is a pointer to the output line (**TPUT**, **TPUT_ASID**, and **TPUT_USERID**) or to the input buffer (**TGET**).

bufsiz
specifies the length of the output line or the length of the input buffer.

options
is a set of option flags. Each macro corresponds to a value of one of the assembler macro parameters. All option flags are defined as macros in **<tput.h>**. Table 4.3 on page 4-29 lists the options that can be used in each macro. Use **0** as the option argument to specify that all of the defaults should be taken.

asid
is the address space identifier of the target userid.

userid
is a pointer to an 8-byte **char** array that contains the target userid. The userid must be uppercase, left-adjusted, and padded on the right with blanks if necessary.

If the macro name **_AMODE31** has been defined prior to including the **<tput.h>** header file, the macros generate a call to the **_TPUT** function.

RETURN VALUE

The macros return the value in register 15 after the SVC 93 completes.

ERRORS AND DIAGNOSTICS

The possible errors are the same as when the TPUT or TGET Assembler macros are used in an assembler language program.

TPUT/TGET Terminal I/O via SVC 93

(continued)

CAUTIONS

The macros do not perform any error checking on their arguments. Incorrect arguments such as invalid combinations of option flags are either ignored entirely or remain undetected until execution time, at which point they cause unpredictable and possibly undesirable results.

PORTABILITY

None of the macros are portable.

IMPLEMENTATION

By default, the macros generate R-form invocations of TPUT and TGET. This form is not supported by MVS/XA in 31-bit addressing mode.

This function copies output data to a below-the-line buffer before transmission (for TPUT) or reads input data into a below-the-line buffer and then copies it to the buffer pointed to by **`buffer`** (for TGET).

If **`_AMODE31`** has been defined, the maximum transmission size is 1024 bytes. This limit can be changed by recompiling L$UTPIO.

SEE ALSO

See *TSO/E Programming Services.*

EXAMPLE

```
   /* Send greetings.  Use the default options. */
rc = TPUT("Hello, world!",13,0);
```

Table 4.3 on page 4-29 shows the macro names defined as option flags for the **`TPUT`** and **`TGET`** macros. In the macro columns, a yes entry indicates that the flag can be used in the *options* argument for the macro. A no entry indicates that the flag cannot be used. An ignored entry indicates that the flag is ignored if used.

TPUT/TGET Terminal I/O via SVC 93

(continued)

Table 4.3
Macros Defined for TPUT and TGET Options

Option	TPUT	TPUT_ASID	TPUT_USERID	TGET
_EDIT	yes	yes	yes	yes
_ASIS	yes	yes	yes	yes
_CONTROL	yes	yes	yes	no
_FULLSCR	yes	yes	yes	no
_WAIT	yes	yes	yes	yes
_NOWAIT	yes	yes	yes	yes
_HOLD	yes	ignored	ignored	no
_NOHOLD	yes	ignored	ignored	no
_BREAKIN	yes	yes	yes	no
_NOBREAK	yes	yes	yes	no
_HIGHP	ignored	yes	yes	no
_LOW	ignored	yes	yes	no

TTIMER Test or Cancel a Timer Interval

SYNOPSIS

```
#include <spetask.h>

int TTIMER(cancel, unit, void *area);
```

DESCRIPTION

The **TTIMER** macro implements the functionality of the MVS Assembler TTIMER macro. This macro is supported only with the Systems Programming Environment (SPE).

The **cancel** argument indicates whether the timer interval is to be cancelled or merely tested and must be either **CANCEL** or **NOCANCEL**. The **unit** argument specifies the units in which remaining time should be stored and must be either **TU** or **MIC**. The **addr** argument addresses an area where the remaining time should be stored. The size of this area is determined by the unit specification, as described in the IBM publication *Application Development Reference: Services for Assembler Language Programs.*

RETURN VALUE

TTIMER returns the value that is returned in register 15 by the assembler TTIMER macro.

CAUTIONS

Use of the **TTIMER** macro in code that uses the full run-time library interferes with library timer handling.

Similar functionality to **TTIMER** can be obtained with the full library using the functions **alarm**, **sleep**, or **clock**.

RELATED FUNCTIONS

STIMER, **STIMERM**, **alarm**, **clock**, **sleep**

typlin Invoke the CMS TYPLIN Function

SYNOPSIS

```
#include <wrterm.h>

int typlin(char *buffer, short bufsiz, char color, char edit);
```

DESCRIPTION

typlin invokes the CMS TYPLIN function to type a line of output to the terminal. The **typlin** macro creates a TYPLIN parameter list and generates an inline SVC 202. A description of each of the arguments follows:

buffer
is a pointer to the string to be typed.

bufsiz
is the length of the string.

color
is one of two values specifying the color in which the string will be typed if the terminal is a typewriter terminal and the typewriter has a two-color ribbon. Use either **WR_RED** or **WR_BLACK**, both of which are defined as macros in **<wrterm.h>**.

edit
is a code specifying the editing to be performed by the TYPLIN function before the string is displayed. Valid values for this argument are defined as macros in **<wrterm.h>**. Each macro corresponds to a value for the EDIT keyword operand in the WRTERM assembler language macro. The macros **WR_EDIT_YES**, **WR_EDIT_NO**, and **WR_EDIT_LONG** correspond to EDIT=YES, EDIT=NO, and EDIT=LONG, respectively.

RETURN VALUE

typlin returns the value in register 15 when the TYPLIN function returns.

CAUTIONS

In XA CMS or ESA, **typlin** can be used only in 24-bit addressing mode.

ERRORS AND DIAGNOSTICS

The possible errors are the same as when the TYPLIN function is used in an assembler language program.

PORTABILITY

typlin is not portable.

IMPLEMENTATION

typlin creates a TYPLIN parameter list in the CRABTAUT work area. See *SAS/C Compiler and Library User's Guide* for more information about CRABTAUT.

typlin Invoke the CMS TYPLIN Function
(continued)

SEE ALSO

See *VM/SP CMS Command Reference.*

EXAMPLE

```
void dispval(int n)
   {
   char *line[130]  ;
   int len, rc;
      /* Format and display the value of the input parameter. */
   len = format(line,"The value of n is %d",n);
   (void) typlin(line,len,WR_BLACK,WR_EDIT_NO);
   }
```

WAIT Wait for the POST of One or More ECB's

SYNOPSIS

```
#include <ostask.h>

void WAIT1(unsigned *ecbaddr);

void WAITM(int n, unsigned **ecblist);
```

DESCRIPTION

The SAS/C **WAIT** macros implement the functionality of the MVS Assembler WAIT macro.

WAIT1 waits for a single ECB to be posted. The **ecbaddr** argument addresses the ECB.

WAITM waits for one or more of a list of ECBs to be posted. The argument **n** to **WAITM** specifies the number of ECBs which must be posted before execution resumes. The argument **ecblist** points to a list of ECB addresses to be waited on. The last address in the list must be flagged by turning on the high-order bit of the address.

CAUTION

WAIT1 and **WAITM** can be used in either an SPE environment or a full run-time library environment. However, when using the full library, you should consider using **ecbsuspend** to wait for an ECB POST because this also allows the wait to be interrupted by a C signal.

EXAMPLE

```
#include <ostask.h>

unsigned myecb = 0, yourecb = 0, theirecb = 0;

unsigned *ecblist[3];

ecblist[0] = &myecb;
ecblist[1] = &yourecb;
ecblist[2] = (unsigned *) (0x80000000 | (unsigned) &theirecb));
WAITM(1, ecblist);                  /* Wait on one of three ECBs. */
```

RELATED FUNCTIONS

ecbsuspend, **POST**

waitrd Invoke the CMS WAITRD Function

SYNOPSIS

```
#include <wrterm.h>

int waitrd(char *inbuf, short inbufsz, char code1, char code2,
           char *pbuf, int pbufsz, int *inlen);
```

DESCRIPTION

waitrd invokes the CMS WAITRD function to read a line of input from the program stack or terminal input buffer. The **waitrd** macro creates a WAITRD parameter list and generates an inline SVC 202. A description of each of the arguments follows:

inbuf
is a pointer to a buffer into which the input line is read.

inbufsz
is the length of the input buffer.

code1
specifies the processing performed on lines read from the terminal input buffer. **<wrterm.h>** contains definitions for a number of macros, each of which corresponds to a WAITRD function code1 parameter. Table 4.4 on page 4-35 lists the macro names and the associated code1 values.

code2
specifies additional processing codes. Again, **<wrterm.h>** contains definitions for a number of macros, each of which corresponds to a WAITRD function code2 parameter. Table 4.5 on page 4-36 lists the macro names and the associated code2 values. For an explanation of each of the code1 and code2 values, refer to Chapter 1 in *VM/SP Command and Macro Reference* appropriate for your release.

pbuf
is a pointer to a prompt string. If **code2** is PROMPT_NO, this argument is ignored.

pbufsz
is the length of the prompt string. If **code2** is PROMPT_NO, this argument is ignored.

inlen
is a pointer to an **int** where the length of the input string is stored.

RETURN VALUE

waitrd returns the value in register 15 when the WAITRD function returns.

CAUTION

In XA CMS or ESA, **waitrd** can only be used in 24-bit addressing mode.

ERRORS AND DIAGNOSTICS

The possible errors are the same as when the WAITRD function is used in an assembler language program.

waitrd Invoke the CMS WAITRD Function
(continued)

PORTABILITY

waitrd is not portable.

IMPLEMENTATION

The **waitrd** macro creates a WAITRD parameter list in the CRABTAUT work area. (See the *SAS/C Compiler and Library User's Guide* for more information about CRABTAUT.)

EXAMPLE

```
   /* Read an input line of at most 25 characters */
   /* from the terminal.                          */
char *name[25] ;
char *prompt = "Please enter your name."
int rc;
int inlen;

   /* Ask for the input to be  */
   /* translated to uppercase. */
rc = waitrd(name,sizeof(name),STACK_UPCASE,PROMPT_YES,
     prompt, strlen (prompt),&inlen);
```

SEE ALSO

See *VM/SP CMS Command Reference.*

Table 4.4
Macros Defined for the waitrd code1 Argument

Macro Name	Corresponding Value
RD_EDIT_YES	U
RD_EDIT_NO	T
RD_EDIT_PHYS	X
RD_EDIT_PAD	S
RD_EDIT_UPCASE	V
RD_EDIT_NOINPT	Y
STACK_UPCASE	Z
STACK_MIXED	W
ATTREST_YES	*
ATTREST_NO	$

waitrd Invoke the CMS WAITRD Function
(continued)

Table 4.5
Macros Defined for the waitrd code2 Argument

Macro Name	Corresponding Value
PROMPT_DIRECT	B
TYPE_DIRECT	D
PROMPT_YES	P
PROMPT_NO	0

WAITT Simulate the WAITT Assembler Language Macro

SYNOPSIS

```
#include <wrterm.h>

void WAITT(void);
```

DESCRIPTION

WAITT simulates the CMS WAITT assembler language macro. **WAITT** creates a CMS CONWAIT function parameter list and generates an inline SVC 202. If the symbol **BIMODAL** is defined, SVC 204 is used in place of SVC 202, which is necessary for use in 31-bit addressing mode.

RETURN VALUE

WAITT does not return a value.

PORTABILITY

WAITT is not portable.

SEE ALSO

See *VM/SP CMS Command Reference.*

WRTERM Simulate the WRTERM Assembly Language Macro

SYNOPSIS

```
#include <wrterm.h>

int WRTERM(char *buffer, short bufsiz);
```

DESCRIPTION

WRTERM simulates the WRTERM assembly language macro. The **WRTERM** macro invokes the **typlin** macro to produce a parameter list that is equivalent to using the WRTERM assembly language macro with all parameters allowed to default. If the symbol **BIMODAL** is defined, **WRTERM** generates an equivalent to the CMS **LINEWRT** macro and, therefore, can be used in 31-bit addressing mode.

RETURN VALUE

WRTERM returns the value in register 15 when the TYPLIN or LINEWRT function returns.

ERRORS AND DIAGNOSTICS

The possible errors are the same as when the WRTERM or LINEWRT macro is used in an assembly language program.

PORTABILITY

WRTERM is not portable.

IMPLEMENTATION

The definition of **WRTERM** if **BIMODAL** is not defined is as follows:

```
#define WRTERM(bf,l) typlin(bf,l,WR_BLACK,WR_EDIT_NO)
```

SEE ALSO

See *VM/ESA CMS Application Development Reference for Assembler*.

EXAMPLE

```
void dispval(int n)
   {
   char *line[130]  ;
   int len, rc;

      /* equivalent to the example for the typlin macro */
   len = format(line,"The value of n is %d",n);
   (void) WRTERM(line,len);
   }
```

5 Inter-User Communications Vehicle (IUCV) Functions

5-1 Introduction
5-1 IUCV Communications Overview
5-3 IUCV, Signal Handling, and Message Processing
5-5 IUCV Parameter Lists and External Interrupt Data Formats
5-7 Message Queues
5-8 Return Codes
5-8 Function Descriptions
5-15 An Example of IUCV Communication
5-16 The SENDER program
5-18 The RECEIVER program
5-21 Guidelines and Cautions

Introduction

The SAS/C Library defines the SIGIUCV asynchronous signal in support of programs that want to take advantage of IUCV communications. The library also provides a set of functions that invoke VM IUCV functions and the CMS support of IUCV. The SAS/C functions, when used in conjunction with the SIGIUCV signal, enable programs running in CMS to communicate with other virtual machines, CP system services, or themselves.

This chapter covers the SAS/C IUCV functions. Topics include a brief introduction to IUCV communications in a virtual machine, CMS support of IUCV, IUCV parameter lists and external interrupt data formats, and IUCV library functions.

Although an overview of the IUCV and CMS IUCV functions is provided, this discussion of IUCV communications is written primarily for programmers experienced with IUCV. A knowledge of CP and CMS is assumed. More detailed coverage of all aspects of IUCV can be found in the appropriate IBM publications for your release of VM. For a complete description of how the SAS/C Library supports asynchronous signals and the use of signal-related SAS/C functions, refer to Chapter 5, "Signal-Handling Functions," in *SAS/C Library Reference, Volume 1.*

IUCV Communications Overview

IUCV is a communications facility that enables programs running in a virtual machine to communicate with programs in other virtual machines. An IUCV program also can send data to or receive data from a CP system service or communicate asynchronously with itself. IUCV allows any amount of data to be transferred during a single transaction.

IUCV transfers data in blocks known as *messages.* Messages travel from a source communicator to a target communicator along a route called the *message path.* Each communicator can have many paths and can be a source communicator on some paths and a target communicator on others simultaneously.

When you write a program that permits IUCV communication, you invoke IUCV functions to perform the following basic IUCV tasks:

- initialize IUCV communications
- connect to another virtual machine
- accept a connection with a virtual machine
- send, receive, and reply to IUCV messages
- sever a connection
- terminate IUCV communications.

Additional IUCV functions also perform these tasks:

- reject an IUCV message
- test a message for completion
- purge a message
- quiesce (temporarily suspend) an IUCV path
- resume a communication on a quiesced path.

IUCV recognizes no entity smaller than a virtual machine. In order for several programs in a CMS virtual machine to use IUCV at the same time, CMS provides supporting IUCV routines that manage IUCV paths and route messages to the responsible programs. CMS IUCV maintains a table of IUCV programs and associates a name (provided by the program) with each IUCV path active in the virtual machine.

Programs written in assembler invoke IUCV functions via an instruction known as the *IUCV macro instruction.* The IUCV macro instruction specifies the IUCV function requested and a parameter list that contains the information needed to perform it. CMS IUCV services are provided by two assembler macros called HNDIUCV and CMSIUCV. HNDIUCV provides support for IUCV initialization and termination. CMSIUCV handles the connection to the communicating program. The CMSIUCV and HNDIUCV macros require that the program provide a name for CMS to use to keep track of IUCV paths. If a CMS IUCV macro in turn invokes an IUCV macro instruction, an IUCV parameter list also is required.

All SAS/C IUCV functions invoke either the IUCV macro instruction or one of the CMS IUCV macros. When a SAS/C function uses CMS IUCV support, the program must provide a name parameter and possibly an IUCV parameter list. When a SAS/C function invokes the IUCV macro instruction directly, the IUCV parameter list must be provided.

The SAS/C functions are described in detail in the following pages, including specification of their parameters and the CMS IUCV service or IUCV macro invoked. The **`<cmsiucv.h>`** header file, providing structure definitions for the IUCV parameter lists and external interrupt data, also is described.

To summarize the IUCV functions and tasks involved, the following table shows IUCV functions as implemented by the library. Listing the macro and parameter involved in the function call illustrates how the library interfaces with IUCV and CMS.

SAS/C Function	Purpose	IUCV or CMS Macro
`iucvset`	Initialize IUCV communications	Invoke HNDIUCV macro with SET parameter
`iucvacc`	Accept a pending connection	Invoke CMSIUCV macro with ACCEPT parameter
`iucvconn`	Request that a path be established	Invoke CMSIUCV macro with CONNECT parameter
`iucvsevr`	Terminate an existing path	Invoke CMSIUCV macro with SEVER parameter
`iucvclr`	Terminate IUCV communications	Invoke HNDIUCV macro with CLR parameter

continued

SAS/C Function	Purpose	IUCV or CMS Macro
iucvrply	Respond to a message	Invoke the IUCV macro instruction for REPLY
iucvrecv	Accept a message	Invoke the IUCV macro instruction for function RECEIVE
iucvrej	Refuse a message	Invoke the IUCV macro instruction for function REJECT
iucvsend	Send a message	Invoke the IUCV macro instruction for function SEND
iucvqs	Temporarily suspend communication	Invoke the IUCV macro instruction for function QUIESCE
iucvresm	Restore communication after a QUIESCE	Invoke the IUCV macro instruction for function RESUME
iucvtcmp	Determine if a message has been completed	Invoke the IUCV macro instruction for function TEST COMPLETION
iucvpurg	Terminate a message	Invoke the IUCV macro instruction for function PURGE

IUCV, Signal Handling, and Message Processing

Inter-user communication is driven by asynchronous IUCV external interrupts. That is, IUCV requires synchronization and cooperation of communicating processes. As seen in the previous section, the library relies on IUCV and CMS support of IUCV to help coordinate these interrupts. However, programs initiate and handle IUCV interrupts in the context of SAS/C signal processing. All IUCV interrupts are identified by the asynchronous signal SIGIUCV. A program must first call **signal** or **sigaction** to establish a handler for SIGIUCV signals. Calling **signal** identifies a function (called a handler) that should be invoked when an IUCV interrupt occurs. The handler should call the signal function **siginfo**, which returns a pointer to a structure containing the external interrupt data.

Signal processing handles the IUCV interrupts associated with the basic IUCV tasks. IUCV external interrupts can be divided into those concerning connections and paths and those prompted by message transfer. The IUCV external interrupt types are summarized in the following table.

Connection and Path Interrupts	Message Transfer Interrupts
Connection pending	Incoming priority message
Connection complete	Incoming message nonpriority
Path severed	Incoming priority reply
Path quiesced	Priority reply
Path resumed	

The library, at the time of the first call to **signal** with the SIGIUCV signal identifier, initializes all library handling of IUCV external interrupts for the program and associates a SIGIUCV handler with the external interrupt. Although **iucvset** can be called any number of times within a program to establish multiple message paths, only one SIGIUCV handler can be associated with IUCV signals at any one time. Of course, programs can redefine handlers in subsequent calls to **signal**, or a single handler can use the external interrupt type to route the signal to subsidiary functions. Because each message is uniquely identified, the SIGIUCV handler can use this data to route signals of similar types to other functions.

A simple example of the complete IUCV communication process is given in the following table. In this example, calls to the signal functions **sigblock**, **sigpause**, and **siginfo** are combined with SAS/C IUCV function calls to process a message sent from a source virtual machine to a receiving or target virtual machine. Note that before any message transfer can occur, a call must be made to **signal** to identify the handler for subsequent SIGIUCV interrupts. Note also that the call to **signal** must be paired with a call to **iucvset** to initialize communication (step 1). These calls must be made by both sender and receiver before any inter-user communication can take place. Subsequent calls are traced back and forth across the table. In step 2, for example, the receiving program issues a call to **sigpause** to wait for the sending program's interrupt. In step 3, the sender issues an **iucvconn** to attempt a connection with the receiver, and so on.

Sending Program Function Called	Purpose of Call	Receiving Program Function Called
1. **signal** **sigblock** **iucvset**	Initialize communications	**signal** **sigblock** **iucvset**
	Wait for interrupt	2. **sigpause**
3. **iucvconn**	Sender attempts to connect; receiver recognizes a connection pending	4. **siginfo**
5. **sigpause**	Sender waits until receiver accepts connection	6. **iucvacc**

continued

Sending Program Function Called	Purpose of Call	Receiving Program Function Called
7. **siginfo** **iucvsend** **sigpause**	Sender recognizes connection is complete, sends a message, and waits for the reply	
	Receiver recognizes an incoming message, receives it, replies, and waits	8. **siginfo** **iucvrecv** **iucvrply** **sigpause**
9. **siginfo**	Sender recognizes an incoming reply and processes it	
10.**iucvsevr** **iucvclr**	Message transfer complete; sender severs path and clears; receiver recognizes path severed	11. **siginfo**

In this table, many of the IUCV function calls occur based on the information returned from the call to **siginfo**. This signal function returns a pointer to one of three external interrupt structures. The structure pointed to depends on the interrupt subtype. These structures and their parameters are easily accessible to the program. The next section discusses the structures involved in IUCV communication and the information available to the program.

IUCV Parameter Lists and External Interrupt Data Formats

The header file for the IUCV functions is **<cmsiucv.h>**. Three structures cover all IUCV parameter lists and external interrupt data formats. Each structure used in IUCV communication is discussed on the following pages. As indicated by the comments, **<cmsiucv.h>** parameter list fields correspond to fields in the parameter lists for each individual IUCV function. These fields are explained in detail in the IBM documentation mentioned in “Introduction” on page 5-1 .

The structure **iucv_path_plist** defines the ACCEPT, CONNECT, QUIESCE, RESUME, and SEVER parameter list. External interrupt data are defined for PENDING CONNECTION, CONNECTION COMPLETE, SEVER, QUIESCE, and RESUME external interrupts.

```
struct iucv_path_plist {
   short pathid;                /* IPPATHID                    */
   char flags1;                 /* IPFLAGS1                    */
   union {
      char rcode;               /* IPRCODE                     */
      char type;                /* IPTYPE                      */
   } ip;
   short msglim;                /* IPMSGLIM                    */
   char fcncd;                  /* IMFCNCD                     */
   char _;
   char vmid[8];                /* IPVMID                      */
   char pgm[8];                 /* IPUSER                      */
```

```
   char user[8];
   double _d;                    /* (used only for alignment)     */
};

typedef struct iucv_path_plist iucv_path_data;
```

The structure **iucv_msg_plist** defines the parameter list for the RECEIVE, REJECT, REPLY, and SEND functions. External interrupt data are defined for the pending message external interrupt.

```
struct iucv_msg_plist {
   short pathid;                 /* IPPATHID                      */
   char flags1;                  /* IPFLAGS1                      */
   union {
      char rcode;                /* IPRCODE                       */
      char type;                 /* IPTYPE                        */
   } ip;
   int msgid;                    /* IPMSGID                       */
   int tgrcls;                   /* IPTRGCLS                      */
   union {
      char data[8];              /* IPRMMSG1/IPRMMSG2             */
      struct _adlen bf1;         /* IPBFADR1/IPBFLN1F             */
   } msg;
   int srccls;                   /* IPSRCCLS                      */
   int msgtag;                   /* IPMSGTAG                      */
   struct _adlen bf2;            /* IPBFADR2/IPBFLN2F             */
   double _d;                    /* (used only for alignment)     */
};

typedef struct iucv_msg_plist iucv_msg_data;
```

The last of the three IUCV structures is **iucv_comp_plist**. This structure defines the parameter list for the PURGE and TEST COMPLETION functions. External interrupt data are defined for the complete message external interrupt.

```
struct iucv_comp_plist {
   short pathid;                 /* IPPATHID                      */
   char flags1;                  /* IPFLAGS1                      */
   union {
      char type;                 /* IPTYPE                        */
      char rcode;                /* IPRCODE                       */
   } ip;
   int msgid;                    /* IPMSGID                       */
   unsigned audit;               /* IPAUDIT (byte 4 not applicable)*/
   char msgdata[8];              /* IPRMMSG1/IPRMMSG2             */
   int srccls;                   /* IPSRCCLS                      */
   int msgtag;                   /* IPMSGTAG                      */
   int _;
   int bfln2f;                   /* IPBFLN2F                      */
   double _d;                    /* (used only for alignment)     */
};

typedef struct iucv_comp_plist iucv_comp_data;
```

The values for the interrupt subtypes, as defined for the structure variable **ip.type**, are shown in the following table:

Variable	Value
CONNECTION_PENDING	1
CONNECTION_COMPLETE	2
PATH_SEVERED	3
PATH_QUIESCED	4
PATH_RESUMED	5
INCOMING_PRI_REPLY	6
INCOMING_REPLY	7
INCOMING_PRI_MSG	8
INCOMING_MSG	9

The type definitions (such as **iucv_path_data**) and interrupt types (**ip.type**) enable you to control the way your program handles IUCV interrupts. The following code illustrates how this can be done:

```
iucv_path_data *XID;

XID = (iucv_path_data *) siginfo();
switch (XID->ip.type) {
   case CONNECTION_COMPLETE:
      break;
   case INCOMING_REPLY:   /* Get reply length. */
      rep_len = ((iucv_msg_data *)XID)->bf2.ln;
      break;
}
```

In this section of code, a cast expression converts the value returned by **siginfo** to a pointer of type **iucv_path_data** so it can be assigned to XID. A case statement is then used to handle the possible interrupt subtypes returned in **ip.type**.

Message Queues

When an IUCV interrupt occurs, the library places the IUCV external interrupt data on a *pending interrupt data* queue until it is possible to raise the SIGIUCV signal. Refer to the section “Discovering Asynchronous Signals” in Chapter 5 of *SAS/C Library Reference, Volume 1.* There is no limit on the number of interrupts that can be placed on the queue.

Return Codes

The library also interfaces with IUCV and CMS in providing return codes. The return codes operate as follows:

- A negative return code indicates that the library could not invoke a CMS IUCV function.
- Library functions that successfully invoke CMS IUCV return the CMS return code. (CMS uses 0 to indicate success.)
- Library functions that invoke IUCV macro instructions directly return the condition code set by IUCV.

For a complete list and explanation of IUCV function return codes, refer to the *VM/SP System Programmer's Guide* appropriate for your release.

Function Descriptions

Descriptions of each IUCV function follow. 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. A comprehensive example of IUCV communication as implemented by the SAS/C Library follows the function descriptions.

iucvacc Perform an IUCV ACCEPT

SYNOPSIS

```
#include <cmsiucv.h>

int iucvacc(const char *name, struct iucv_path_plist *acc_parmlst);
```

DESCRIPTION

iucvacc requests CMS to issue an IUCV ACCEPT. **name** points to a string that must match the **name** argument to a previously invoked **iucvset** function. **acc_parmlst** is a pointer to an IUCV ACCEPT parameter list.

RETURN VALUE

iucvacc returns **0** if the function call was successful. If **signal(SIGIUCV,fp)** has never been called, **iucvacc** returns **–1**. If the CMSIUCV macro returns a nonzero return code, **iucvacc** returns that value. If an IUCV error occurs, **iucvacc** returns **1**, and the value of IPRCODE is available in **acc_parmlst->ip.rcode**.

CAUTION

You must call **iucvset** before calling **iucvacc**.

IMPLEMENTATION

iucvacc invokes the CMS CMSIUCV macro with the ACCEPT parameter.

EXAMPLE

```
#include <cmsiucv.h>

   /* Issue an IUCV ACCEPT in response to a             */
   /* pending connection.                               */
int rc;
   /* external interrupt data for a pending connection  */
iucv_path_data *p_connect;
   /* other IUCV CONNECT parameter list initializations */
.
.
.
p_connect->ip.type = 0;

rc = iucvacc("RECEIVER",p_connect);
```

iucvclr Terminate IUCV Communications

SYNOPSIS

```
#include <cmsiucv.h>

int iucvclr(const char *name);
```

DESCRIPTION

iucvclr terminates IUCV communications for a program. **name** points to a string that must match the **name** argument to a previously invoked **iucvset** function.

RETURN VALUE

iucvclr returns **0** if the function call was successful. If **signal(SIGIUCV,fp)** has not been called, **iucvclr** returns **−1**. If the HNDIUCV macro returns a nonzero return code, **iucvclr** returns that value. If an IUCV error occurs, **iucvclr** returns **1000** + the value of IPRCODE (as does HNDIUCV).

CAUTION

You must call **iucvset** before calling **iucvclr**.

IMPLEMENTATION

iucvclr invokes the CMS HNDIUCV macro with the CLR parameter and frees the external interrupt data queue associated with **name**.

EXAMPLE

```
#include <cmsiucv.h>
.
.
.
rc = iucvclr("SENDER");
```

iucvconn Perform an IUCV CONNECT

SYNOPSIS

```
#include <cmsiucv.h>

int iucvconn(const char *name, struct iucv_path_plist *conn_parmlst);
```

DESCRIPTION

iucvconn requests CMS to issue an IUCV CONNECT. **name** points to a string that must match the **name** argument to a previously invoked **iucvset** function. **conn_parmlst** is a pointer to an IUCV CONNECT parameter list.

RETURN VALUE

iucvconn returns **0** if the function call was successful. If **signal(SIGIUCV,fp)** has not been called, **iucvconn** returns **–1**. If the CMSIUCV macro returns a nonzero return code, **iucvconn** returns that value. If an IUCV error occurs, **iucvconn** returns **1**, and the value of IPRCODE is available in **conn_parmlst->ip.rcode**.

CAUTION

You must call **iucvset** before calling **iucvconn**.

IMPLEMENTATION

iucvconn invokes the CMS CMSIUCV macro with the CONNECT parameter.

EXAMPLE

```
#include <cmsiucv.h>
#include <lcstring.h>

int rc;
struct iucv_path_plist path;
.
.
.

   /* Name the receiving userid and program. */
memset((char *) &path,0,sizeof(path));
memcpyp(path.vmid,"USERID",sizeof(path.vmid),6,' ');
memcpy(path.pgm,"RECEIVER",8);

rc = iucvconn("SENDER",&path);
```

IUCV Perform IUCV Functions

SYNOPSIS

```
#include <cmsiucv.h>

int iucvpurg(struct iucv_comp_plist *comp_p);
int iucvqs(struct iucv_path_plist *path_p);
int iucvrecv(struct iucv_msg_plist *msg_p);
int iucvrej(struct iucv_msg_plist *msg_p);
int iucvresm(struct iucv_path_plist *path_p);
int iucvrply(struct iucv_msg_plist *msg_p);
int iucvsend(struct iucv_msg_plist *msg_p);
int iucvtcmp(struct iucv_comp_plist *comp_p);
```

DESCRIPTION

Each of these functions performs an IUCV function. **msg_p**, **path_p**, and **comp_p** point to the IUCV parameter list associated with that function. The function names and IUCV functions are associated as follows:

SAS/C function	IUCV function
iucvpurg	PURGE
iucvqs	QUIESCE
iucvrecv	RECEIVE
iucvrej	REJECT
iucvresm	RESUME
iucvrply	REPLY
iucvsend	SEND
iucvtcmp	TEST COMPLETION

RETURN VALUE

All of the functions return the condition code set by the IUCV macro instruction.

CAUTION

You must have established an IUCV signal handler with **signal** or **sigaction** and initialized IUCV communications with **iucvset** before invoking any of these functions.

IMPLEMENTATION

The functions invoke the IUCV macro with the associated function name and parameter list.

EXAMPLE

See the example for **iucvset**.

IUCV Perform IUCV Functions
(continued)

RELATED FUNCTIONS

`sigaction`, **`signal`**

SEE ALSO

Chapter 5, "Signal-Handling Functions," in *SAS/C Library Reference, Third Edition, Volume 1.*

iucvset Initialize IUCV Communications

SYNOPSIS

```
#include <cmsiucv.h>

int iucvset(const char *name, int *maxconns);
```

DESCRIPTION

iucvset identifies an IUCV program to CMS. **name** is a 1- to 8-character string identifying the communicating program's name. For example, a SAS/C program called **sender.c** may specify the name **''SENDER''**. If **name** is less than eight characters long, it is padded with blanks on the right. If **name** is greater than eight characters, it is truncated. The maximum number of connections for the virtual machine is returned in the integer pointed to by **maxconns**.

You can call **iucvset** any number of times as long as each call uses a different **name** string.

RETURN VALUE

iucvset returns **0** if the function call was successful. **iucvset** returns a negative code as follows:

- **−1** if **signal** or **sigaction** has never been called to handle the SIGIUCV signal.
- **−2** if a previous call to **iucvset** had the same **name** parameter.
- **−3** if there is not enough memory to allocate an interrupt queue.

If the HNDIUCV macro returns a nonzero return code, **iucvset** returns that value.

CAUTIONS

You must call **signal** or **sigaction** to establish an IUCV interrupt handler before calling **iucvset**. An IUCV program must be ready to handle incoming IUCV interrupts even before **iucvset** has returned.

IMPLEMENTATION

iucvset invokes the CMS HNDIUCV macro with the SET parameter. In addition, it allocates an initial 4K external interrupt data buffer.

EXAMPLE

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

int maxconns, rc;
.
.
.

rc = iucvset("SENDER",&maxconns);
if (rc != 0)
   puts("iucvset failed.");
```

iucvsevr Perform an IUCV SEVER

SYNOPSIS

```
#include <cmsiucv.h>

int iucvsevr(const char *name, struct iucv_path_plist *svr_parmlst,
             const char *code);
```

DESCRIPTION

iucvsevr requests CMS to issue an IUCV SEVER. **name** points to a string that must match the **name** argument to a previously invoked **iucvset** function. **svr_parmlst** is a pointer to an IUCV SEVER parameter list.

code can be either **''ALL''** or **''ONE''**. If it is **''ONE''**, only the one path specified by **svr_parmlist.pathid** is severed (using SEVER). If **code** is **''ALL''**, all paths owned by the program are severed. The **''ALL''** and **''ONE''** parameters must be coded exactly as shown, uppercase and enclosed in quotation marks, with no substitutions or variations in syntax.

RETURN VALUE

iucvsevr returns **0** if the function call was successful. If neither **signal** nor **sigaction** has not been called to handle the SIGIUCV signal, **iucvsevr** returns **-1**. If the CMSIUCV macro returns a nonzero return code, **iucvsevr** returns that value. If an IUCV error occurs, **iucvsevr** returns **1**, and the value of IPRCODE is available in **svr_parmlst->ip.rcode**.

CAUTION

You must call **iucvset** before calling **iucvsevr**.

IMPLEMENTATION

iucvsevr invokes the CMS CMSIUCV macro with the SEVER parameter. The value of the CODE parameter depends on the value of **code**.

EXAMPLE

```
#include <cmsiucv.h>

short pathid;           /* PATHID to be severed */

path.pathid = pathid;
rc = iucvsevr("SENDER",&path,"ONE");
```

An Example of IUCV Communication

The following example illustrates how two virtual machines can communicate with each other by sending IUCV messages. The first example program is called SENDER. This program prompts for a message string to be input from the terminal and sends the string to the second example program, RECEIVER. RECEIVER displays the message and prompts for a reply string. The reply string is sent back to SENDER. SENDER displays the reply, and the cycle repeats until a zero-length line is entered in response to SENDER's prompt.

The SENDER program

This program has only two functions. The **main** function initializes signal processing and IUCV. It uses **fgets** to read message strings and **printf** to display reply strings. The **sigpause** function makes the program wait until a reply is sent. The **sigblock** function blocks SIGIUCV signals until the program is ready to handle them.

The **iucvtrap** function is the SIGIUCV signal handler. Only two IUCV interrupt types are expected: a CONNECTION COMPLETE interrupt in response to the IUCV CONNECT and an INCOMING REPLY in response to each IUCV SEND.

The source code for these examples is provided with the compiler and library. Ask your SAS Software Representative for SAS/C software products for information about obtaining copies of these programs.

```
                /* SENDER */

#include <lcsignal.h>
#include <lcstring.h>
#include <cmsiucv.h>
#include <lcio.h>

void iucvtrap(void);                    /* Declare SIGIUCV handler.*/

   /* Set up stdin amparms.*/
char *_stdiamp = "prompt=What message do I send?,eof=";

int rep_len = 0;
short pathid = 0;

main()
{
   struct iucv_path_plist path;    /* IUCV plist for CONNECT, SEVER  */
   struct iucv_msg_plist msg;      /* IUCV plist for SEND            */

      /* maximum number of IUCV connections for this virtual machine */
   int maxconns;
   int rc;                         /* return code from IUCV functions*/
   char message[120],              /* message buffer                 */
   reply[120];                     /* reply buffer                   */

      /* Initialize IUCV signal processing and establish a handler.  */
      /* Block IUCV signals until we're ready to handle them.        */
   signal(SIGIUCV,&iucvtrap);
   sigblock(1 << (SIGIUCV-1));

      /* Identify this program as SENDER. */
   if ((rc = iucvset("SENDER",&maxconns)) != 0) {
      printf("Return code from iucvset was %d\n",rc);
      exit(4);
   }
   printf("Maximum IUCV connections: %d\n",maxconns);
```

```
  /* Fill in the IUCV "path" parameter list with the target      */
  /* userid and the name of the target program. All of the other */
  /* parameters are superfluous in this program.                 */
memset((void *) &path,0,sizeof(path));
memcpy(path.vmid,"SASCUSER",8);
memcpy(path.pgm,"RECEIVER",8);

  /* Request an IUCV CONNECT to the userid/program pair named    */
  /* in the parameter list.  Check for an IUCV error (return     */
  /* code of 1)and print the IUCV error code.                    */
rc = iucvconn("SENDER",&path);
if (rc != 0) {
   printf("Return code from iucvconn was %d\n",rc);
   if (rc == 1)
      printf("IPRCODE = %d\n",path.ip.rcode);
   exit(8);
}

  /* Now we are ready. The first interrupt we receive is the     */
  /* CONNECTION COMPLETE interrupt that occurs when RECEIVER     */
  /* issues an IUCV ACCEPT.                                      */
sigpause(0);

  /* Initialize the SEND parameter list with the pathid and      */
  /* buffer addresses and the length of the reply buffer.        */
memset((void *) &msg,'\0',sizeof(msg));
path.pathid = msg.pathid = pathid;
msg.msg.bf1.adr = message;
msg.bf2.adr = reply;
msg.bf2.ln = sizeof(reply);

  /* Prompt for input messages and send until EOF.               */
fgets(message,sizeof(message),stdin);
while (!feof(stdin)) {
     /* Put message length in the SEND parameter list.          */
   msg.msg.bf1.ln = strlen(message) - 1;
     /* Send message.  Wait (via sigpause) for reply.  Upon     */
     /* receipt of reply, 'rep_len' contains the number of      */
     /* unused bytes in the reply buffer.  Print the reply      */
     /* and prompt for a new message.                           */
   iucvsend(&msg);
   sigpause(0);
   rep_len = sizeof(reply) - rep_len;
   printf("RECEIVER replies \"%.*s\"\n",
           rep_len,reply);
   fgets(message,sizeof(message),stdin);
}
```

```
      /* We are ready to quit. SEVER the IUCV path.  Check for IUCV   */
      /* errors as before.                                            */
   if ((rc = iucvsevr("SENDER",&path,"ONE")) != 0) {
      printf("Return code from iucvsever = %d\n",rc);
      if (rc == 1)
         printf("IPRCODE = %d\n",path.ip.rcode);
      exit(10);
   }

      /* Terminate IUCV communications for SENDER. */
   iucvclr("SENDER");
   exit(0);
}

   /* The SIGIUCV signal handler. Signals are blocked until return.   */
void iucvtrap(void)
{
      /* Pointer to external interrupt data returned by siginfo. The  */
      /* type is arbitrary since it can point to any of the IUCV      */
      /* structures.                                                  */
   iucv_path_data *XID;

      /* Get a pointer to the external interrupt data.  Use the       */
      /* interrupt type to determine what to do.                      */
   XID = (iucv_path_data *) siginfo();
   switch (XID->ip.type) {
      case CONNECTION_COMPLETE:          /* Save the pathid for SEND. */
         pathid = XID->pathid;
         break;

         /* Extract the number of unused characters in the buffer.    */
      case INCOMING_REPLY:
         rep_len = ((iucv_msg_data *)XID)->bf2.ln;
         break;

         /* Handle unexpected termination of RECEIVER.                */
      case PATH_SEVERED:
         puts("Unexpected SEVER!");
         exit(20);

         /* Handle unexpected type of IUCV signal.                    */
      default:
         printf("Unexpected interrupt type %d\n",XID->ip.type);
         fflush(stdout);
         abort();
   }
      /* Reestablish this function as the SIGIUCV signal handler.     */
   signal(SIGIUCV,&iucvtrap);
   return;
}
```

The RECEIVER program

This program defines three functions. The **main** function, again, initializes signal processing and IUCV. **main** calls **sigpause** to wait for the initial CONNECTION PENDING interrupt once and then repeatedly (until all connections are severed) for incoming messages.

The **iucvtrap** function is again the SIGIUCV signal handler. Three types of IUCV interrupts are expected: a CONNECTION PENDING for each sending program, a PATH SEVERED as each sending program finishes, and an INCOMING MSG interrupt for each message sent by a sending program.

Finally, the **rcvrply** function issues an IUCV RECEIVE for each incoming message and prompts for a reply string. It then sends the reply string via an IUCV REPLY.

```
                /* RECEIVER */

#include <lcsignal.h>
#include <lcstring.h>
#include <cmsiucv.h>
#include <lcio.h>

void iucvtrap(void);                  /* SIGIUCV signal handler      */

                                      /* stdin prompt string         */
char *_stdiamp = "prompt=What do I reply to SENDER?";
int connects = 0;                     /* number of connections made  */
static void rcvrply(iucv_msg_data *); /* Declare internal functions. */

void main()
{
   int maxconns;               /* number of IUCV connections allowed */
   int rc;                     /* return code from IUCV functions    */

      /* Initialize IUCV signal processing and establish a handler.  */
      /* Block IUCV signals until we are ready to handle them.       */
   signal(SIGIUCV,&iucvtrap);
   sigblock(1 << (SIGIUCV-1));

      /* Identify this program as RECEIVER. */
   if ((rc = iucvset("RECEIVER",&maxconns)) != 0) {
      printf("Return code from iucvset was %d\n",rc);
      exit(4);
   }
   printf("Maximum IUCV connections: %d\n",maxconns);

      /* This call waits for the initial                             */
      /* CONNECTION PENDING interrupt.                               */
   sigpause(0);

      /* As long as some SENDER is connected,                        */
      /* wait for incoming messages.                                 */
   while (connects > 0) sigpause(0);
```

```
      /* All paths have been terminated. Terminate IUCV processing.  */
   iucvclr("RECEIVER");
   exit(0);
}

   /* SIGIUCV signal handler. Signals are blocked until return.      */
void iucvtrap(void) {
      /* Pointer to external interrupt data returned by siginfo. The */
      /* type is arbitrary because it can point to any of the IUCV   */
      /* structures.                                                  */
   iucv_path_data *XID;
   int rc;                              /* Return code from iucvacc.*/

      /* Get a pointer to the external interrupt data. Use the       */
      /* interrupt type to determine what to do.                     */
   XID = (iucv_path_data *) siginfo();
   switch (XID->ip.type) {
      case CONNECTION_PENDING:          /* Issue ACCEPT.             */
         XID->ip.type = 0;
         if ((rc = iucvacc("RECEIVER",XID)) != 0) {
            printf("Return code from iucvacc = %d\n",rc);
            if (rc == 1)
               printf("IPRCODE = %d\n",XID->ip.rcode);
            exit(8);
         }
            /* Keep track of the number of connections made.         */
         connects++;
         break;

         /* Call function to get message and send reply. */
      case INCOMING_MSG:

         rcvrply((iucv_msg_data *)XID);
         break;

      case PATH_SEVERED:             /* SENDER decided to stop.      */
         if ((rc = iucvsevr("RECEIVER",XID,"ONE")) != 0) {
            printf("Return code from iucvsevr = %d\n",rc);
            exit(8);
         }
         connects--;                 /* Update number of connections.*/
         break;

      default:                       /* Handle other interrupt types.*/
         printf("Unexpected interrupt type %d\n",XID->ip.type);
         fflush(stdout);
         abort();
   }

      /* Re-establish this function as the SIGIUCV signal handler.   */
   signal(SIGIUCV,&iucvtrap);
   return;
}

      /* Function to issue IUCV RECEIVE and print the message.       */
```

```
static void rcvrply(m_data)
iucv_msg_data *m_data;
{
   int msg_len;                          /* incoming message length */
   char msg_buffer[120];                 /* message buffer          */

      /* Create IUCV RECEIVE parameter list using the external       */
      /* interrupt data area.  Issue the IUCV RECEIVE.               */
   m_data->msg.bf1.adr = msg_buffer;
   m_data->msg.bf1.ln = sizeof(msg_buffer);
   iucvrecv(m_data);

      /* Upon return, m_data->msg.bf1.ln contains the number of      */
      /* unused bytes in the message buffer.  Print the message.     */
   msg_len = sizeof(msg_buffer) - m_data->msg.bf1.ln;
   printf("SENDER says \"%.*s\"\n",
        msg_len,msg_buffer);

      /* Prompt for a reply message.  If EOF, quit.                  */
   fgets(msg_buffer,sizeof(msg_buffer),stdin);
   if (feof(stdin)) {
      puts("Terminating due to end-of-file.");
      exit(12);
   }

      /* Fill in IUCV REPLY parameter list with buffer               */
      /* address/length and send REPLY.                              */
   m_data->bf2.adr = msg_buffer;
   m_data->bf2.ln = strlen(msg_buffer) - 1;
   iucvrply(m_data);
}
```

Guidelines and Cautions

For additional information concerning IUCV and CMS IUCV support, refer to your VM system's IBM documentation, which covers many topics relevant to IUCV communication including communication with CP system services, user environments, and specific details on the CMS support of IUCV functions.

In using SAS/C IUCV functions, be aware of the following points:

- You must first call the **signal** or **sigaction** function with the SIGIUCV signal as the first argument. This is necessary to establish signal handling before an IUCV signal occurs. You must do this before any IUCV function is called.
- Each call to **iucvset** creating an IUCV path must be matched with an eventual call to **iucvclr** destroying the path. If, at program termination (either by **return** or **exit**), there are still IUCV paths that have not been destroyed via **iucvclr**, the library issues **iucvclr** calls for these paths. Invoking **iucvclr** also frees the space allocated by **iucvset** for queuing interrupts.
- Interrupts trapped by the library but not handled at program termination disappear with no effect.
- If the value of the **code** argument in the **iucvsevr** function is not **''ALL''**, **''ONE''** is assumed.

- As discussed earlier in the section on signal handling and message processing, you must be prepared to handle IUCV interrupts at all times. Each time a SIGIUCV signal occurs, you must reinstate signal handling before another signal occurs. In other words, the program must be prepared to handle interrupts as soon as **`iucvset`** is called. If there are times when you do not want to be interrupted, use **`sigblock`** and **`sigpause`**.
- The examples in this section show the use of **`sigpause`** to wait for an IUCV signal. The use of **`sigblock`** to block the SIGIUCV signal at other times is critical to these examples. If you do not use **`sigblock`** (or **`sigsetmask`**) to block IUCV signals, an expected signal may be discovered before the call to **`sigpause`**. In this event, **`sigpause`** causes the program to wait for a second signal. Depending on the logic of the program, this can cause it to wait for an indefinitely long period.
- The maximum number of paths available to your program is returned in the integer addressed by the second parameter to **`iucvset`**.
- Although there is no limit on the number of interrupts in the library queue, storage allocated by the library for its queue is not freed until the associated message path is terminated by a call to **`iucvclr`** or by program termination. The library uses approximately 44 bytes per enqueued IUCV message. Queue storage is allocated in 4K blocks. If the library cannot allocate a 4K block to extend the queue, then a user ABEND 1228 is issued.
- None of the IUCV functions issue messages or set the **`errno`** variable. If, during program termination, paths are cleared implicitly by the library and an error occurs, an informatory NOTE is issued. However, the program cannot be informed that an error has occurred.

For more information on handling program interrupts and communications, refer to Chapter 5 of *SAS/C Library Reference, Volume 1.*

6 Advanced Program-to-Program Communication/Virtual Machine (APPC/VM) Functions

6-1 Introduction
6-2 APPC/VM Parameter Lists and External Interrupt Data Formats
6-4 Function Descriptions
6-8 APPC/VM Communication Example Programs
6-8 The APPCSEND Program
6-11 The APPCSERV Program
6-16 Guidelines and Cautions

Introduction

This chapter covers the SAS/C APPC/VM functions. These are a set of SAS/C Library functions that invoke the VM/SP APPC/VM Assembler programming interface. These functions are defined according to Release 5 of VM/SP. These functions also work with Release 6 of VM/SP and Release 1 of VM/ESA. For background information on the VM/SP APPC/VM Assembler programming interface, refer to the appropriate IBM documentation for your VM system. For a complete description of how the SAS/C Library supports asynchronous signals and the use of signal-related SAS/C functions, refer to Chapter 5, "Signal-Handling Functions," in *SAS/C Library Reference, Volume 1.*

Note: SAS/C does not provide its own interface to MVS APPC because the standard APPC interface on MVS can be called directly from SAS/C programs. See *Application Development: Writing Transaction Programs for APPC/MVS* (GC28-1121) for details.

The following table shows the APPC/VM functions as implemented by the library.

Table 6.1
APPC/VM Functions

SAS/C Function	Purpose	APPC/VM Function
`appcconn`	Request that a path be established	Invoke APPC/VM CONNECT function
`appcrecv`	Accept a message	Invoke APPC/VM RECEIVE function
`appcscnf`	Send a confirmation request	Invoke APPC/VM SENDCNF function
`appcscfd`	Send a confirmation response	Invoke APPC/VM SENDCNFD function
`appcsdta`	Send data	Invoke APPC/VM SENDDATA function

continued

Table 6.1 (*continued*)

SAS/C Function	Purpose	APPC/VM Function
appcserr	Send an error message	Invoke APPC/VM SENDERR function
appcsreq	Send a request	Invoke APPC/VM SENDREQ function
appcsevr	Break a path	Invoke APPC/VM SEVER function

APPC/VM Parameter Lists and External Interrupt Data Formats

The header file for the APPC/VM functions is **<cmsappc.h>**. Four types of functions are defined in **<cmsappc.h>** that cover all APPC/VM parameter lists and external interrupt data formats. The types are **ident_parms, struct appc_conn_plist, struct appc_send_plist,** and **appc_conn_data**. These types are defined on the following pages.

As indicated by the comments, **<cmsappc.h>** parameter list fields correspond to fields in the parameter lists for each individual APPC/VM function. For more information on these fields, see the IBM documentation mentioned in "Introduction" on page 6-1 .

ident_parms

The **ident_parms** structure defines the user data field when connecting to ***IDENT**.

```
typedef struct ident_parms {
   char   name[8];                 /* name of resource or gateway    */
   char   fcode;                   /* function code                  */
 #define MANAGE_RESOURCE 0x01      /* Identify a resource/gateway.   */
 #define REVOKE_RESOURCE 0x02      /* Revoke a resource/gateway.     */
   char   flag;                    /* various flags                  */
 #define GLOBAL_RESOURCE 0x80      /* Resource should be global.     */
 #define ALLOW_SECURITY_NONE 0x40  /* Allow SECURITY(NONE) connects. */
 #define REVOKE_GLOBAL   0x80      /* Revoke global/gateway resource.*/
   char   rcode;                   /* return code from IUCV SEVER    */
   char   ntype;                   /* resource/gateway flag          */
 #define RESOURCE_ID     0x00      /* Name is resource id.           */
 #define GATEWAY_ID      0x01      /* Name is gateway id.            */
   int    UNUSED;                  /* unused                         */
} ident_parms;
```

appc_conn_plist

The **appc_conn_plist** structure defines the input and output parameter lists for the **APPC CONNECT** function data for the APPC connection complete external interrupt.

```
struct appc_conn_plist {
   short pathid;                    /* IPPATHID                      */
   char flags1;                     /* IPFLAGS1                      */
   union {
      char rcode;                   /* IPRCODE                       */
      char type;                    /* IPTYPE                        */
   } ip1;
   short code;                      /* IPCODE                        */
   union {
      char whatrc;                  /* IPWHATRC                      */
      char flags2;                  /* IPFLAGS2                      */
   } ip2;
   char sendop;                     /* IPSENDOP                      */
   char vmid[8];                    /* IPVMID                        */
   char resid[8];                   /* IPRESID                       */
   int UNUSED_1;
   struct _adlen bf2;               /* IPBFADR2/IPBFLN2F             */
   int UNUSED_2;
   double _d;                       /* not used - forces alignment   */
};
```

appc_conn_data

The **appc_conn_data** structure describes external interrupt data for the IUCV connection complete external interrupt. The **appc_conn_data** type is declared as follows:

```
typedef struct appc_conn_plist appc_conn_data;
```

appc_send_plist

The **appc_send_plist** structure defines the input and output parameter lists for the APPC SENDCNF, SENCFND, SENDDATA, SENDREQ, RECEIVE, and SEVER functions.

```
struct appc_send_plist {
   short pathid;                    /* IPPATHID                      */
   char flags1;                     /* IPFLAGS1                      */
   union {
      char rcode;                   /* IPRCODE                       */
      char type;                    /* IPTYPE                        */
   } ip1;
   short code;                      /* IPCODE                        */
   union {
      char whatrc;                  /* IPWHATRC                      */
      char flags2;                  /* IPFLAGS2                      */
   } ip2;
   char sendop;                     /* IPSENDOP                      */
   unsigned audit;                  /* IPAUDIT (byte 4 not meaningful)*/
   struct _adlen bf1;               /* IPBFADR1/IPBFLN1F             */
   int UNUSED_1[2];
   struct _adlen bf2;               /* IPBFADR2/IPBFLN2F             */
   int UNUSED_2;
   double _d;                       /* (not used - forces alignment)  */
};
```

The values for the interrupt subtypes, as defined for the structure variable **`ip.type`**, are shown in the following table.

Table 6.2
APPC/VM Interrupt Values

Subtype	Value
APPC_CONNECTION_PENDING	0x81
APPC_CONNECTION_COMPLETE	0x82
APPC_SEVER_INTERRUPT	0x83
APPC_FUNCTION_COMPLETE	0x87
APPC_SENDREQ_INTERRUPT	0x88
APPC_INCOMING_MSG	0x89

Function Descriptions

Descriptions of the APPC/VM functions follow. Each description includes a synopsis, description, discussion of return values and portability issues, and an example. Also errors, cautions, diagnostics, implementation details, and usage notes are included where appropriate. A comprehensive example of APPC/VM communication as implemented by the SAS/C Library and “Guidelines and Cautions” on page 6-16 follows the function descriptions.

appcconn Perform an APPC/VM CONNECT

SYNOPSIS

```
#include <cmsappc.h>

int appcconn(const char *name, struct appc_conn_plist *conn_parmlst,
             void *resv);
```

DESCRIPTION

The **appcconn** function invokes the APPC/VM CONNECT function. **name** matches the name of a previously invoked **iucvset** function. **resv** is reserved and should be set to **NULL**.

RETURN VALUE

The **appcconn** function returns **0** if the function call is successful. If neither **signal** nor **sigaction** has been called, **appcconn** returns **-1**. If the CMSIUCV macro returns a nonzero return code, **appcconn** returns that value. If an IUCV error occurs, **appcconn** returns **1**, and the value of IPRCODE is available in **conn_parmlst->ip.rcode**.

CAUTION

You must call **iucvset** before calling **appcconn**.

EXAMPLE

```
#include <cmsappc.h>
#include <lcstring.h>

int rc;
struct appcc_conn_plist path;
.
.
.

rc = appcconn("APPCSEND",&conn, NULL);
```

RELATED FUNCTIONS

iucvset

appc Perform APPC/VM Functions

SYNOPSIS

```
#include <cmsappc.h>

extern int appcrecv(struct appc_send_plist *send_p);
extern int appcscnf(struct appc_send_plist *send_p);
extern int appcscfd(struct appc_send_plist *send_p);
extern int appcsdta(struct appc_send_plist *send_p);
extern int appcserr(struct appc_send_plist *send_p);
extern int appcsreq(struct appc_send_plist *send_p);
```

DESCRIPTION

Each of these functions performs an APPC/VM function. The function names and APPC/VM functions are associated as follows:

SAS/C Function	APPC/VM Function
appcrecv	RECEIVE
appcscnf	SENDCNF
appcscfd	SENDCNFD
appcsdta	SENDDATA
appcserr	SENDERR
appcsreq	SENDREQ

RETURN VALUE

All of the functions return the condition code set by the APPC/VM macro instruction.

CAUTION

You must establish an IUCV signal handler with either **signal** or **sigaction** and initialize IUCV communications with **iucvset** before invoking any of these functions.

IMPLEMENTATION

The **appc** functions invoke the APPC/VM Assembler interface.

EXAMPLE

See “The APPCSEND Program” on page 6-8 and “The APPCSERV Program” on page 6-11.

RELATED FUNCTIONS

iucvset

appcsevr Perform an APPC/VM SEVER

SYNOPSIS

```
#include <cmsappc.h>

int appcsevr(const char *name, struct appc_send_plist *send_p,
             const char *code);
```

DESCRIPTION

The **appcsevr** function invokes the APPC/VM SEVER function. **name** is the name passed to a previously invoked **appcconn** function; **send_p** is the pointer to an APCC/VM parameter list, and **code** 6 is either **''ALL''** or **''ONE''**. Refer to “iucvsevr” on page 5-15 for more information on the *code* parameter.

RETURN VALUE

The **appcsevr** function returns **0** if the function call is successful. If either **signal** or **sigaction** has not been called, **appcsevr** returns **−1**. If the CMSIUCV macro returns a nonzero return code, **appcsevr** returns that value. If an IUCV error occurs, **appcsevr** returns **1**, and the value of IPRCODE is available in **send_p->ip.rcode**.

CAUTION

You must call **iucvset** before calling **appcsevr**.

IMPLEMENTATION

The **appcsevr** function invokes the APPC/VM SEVER function.

EXAMPLE

```
#include <cmsappc.h>

short pathid;           /* PATHID to be severed */

path.pathid = pathid;
rc = appcsevr("SENDER",&path,"ONE");
```

APPC/VM Communication Example Programs

The following example programs illustrate how the APPC/VM interface can be used to communicate with a global resource manager program. The user program is named APPCSEND. The global resource manager program is named APPCSERV. These programs assume that the global resource is within the same TSAF collection.

Start APPCSERV first. When started, the APPCSEND program prompts for a message to be sent to APPCSERV. APPCSERV displays the message and prompts for a reply message. The reply, in turn, is sent back to APPCSEND, which displays the reply and prompts for another message. This cycle continues until APPCSEND gets a null input string.

The APPCSEND Program

This program has two functions: **main**, which contains the bulk of the program code, and **appctrap**, the SIGIUCV signal-handler function. The **main** function initializes IUCV signal processing and requests a connection to the target resource APPCSERV. If the request is granted, APPCSEND prompts the user for a message string and sends the message to APPCSERV. It then waits for a reply that, when received, is displayed on the terminal. If a null message is read, the path to APPCSEND is severed. The **appctrap** accepts the initial CONNECTION COMPLETE interrupt from APPCSERV. This function also handles the FUNCTION COMPLETE and SEVER INTERRUPT external interrupts.

Here is the APPCSEND program:

```
#include <lcsignal.h>
#include <lcstring.h>
#include <cmsappc.h>
#include <lcio.h>

void appctrap(void);                /* Declare SIGIUCV handler.       */

                                    /* Set up stdin amparms.          */
char *_stdiamp = "prompt=What message do I send?\n, eof=";
static int rep_len = 0;
static short pathid = 0;
static short ident_pathid = 0;

main()
{
   struct appc_conn_plist conn;  /* APPC plist for CONNECT, SEVER     */
   struct appc_send_plist send;  /* APPC plist for SENDDATA, SENDCNFD */
   int maxconns,                 /* maximum number of IUCV connec-    */
                                 /* tions for this virtual machine    */
       rc;                       /* return code from APPC functions   */
   struct senddata {
      unsigned short length;     /* length of data to send            */
      char message[120+2];       /* message buffer                    */
   } logrec, reply;              /* log record and reply buffers      */

      /* Initialize IUCV signal processing and establish a handler.   */
      /* Block IUCV signals until we're ready to handle them.         */
   signal(SIGIUCV,&appctrap);
   sigblock(1 << (SIGIUCV-1));
```

```
   /* Identify this program as APPCSEND.                          */
if ((rc = iucvset("APPCSEND",&maxconns)) != 0) {
   printf("Return code from iucvset was %d\n", rc);
   exit(4);
}
printf("Maximum IUCV connections: %d\n", maxconns);

   /* Fill in the APPC "conn" parameter list with the target      */
   /* resource id, "APPCSERV".                                    */

   /* Note that the resource name in IPRESID must be eight bytes  */
   /* long, padded on the right with blanks if necessary.         */
memset(&conn, 0, sizeof(conn));
conn.ip2.flags2 = IPLVLCF + IPMAPPED;
memcpy(conn.resid, "APPCSERV",8);

   /* Request an APPC CONNECT to the resource named in the        */
   /* parameter list.  Check for an IUCV error (return code != 0) */
   /* and print the APPC error code.                              */
rc = appcconn("APPCSEND", &conn, 0);
if (rc != 0) {
   printf("Return code from appcconn was %d\n", rc);
   if (rc == 1)
      printf("IPRCODE = x'%X'\n", conn.ip1.rcode);
   exit(8);
}

   /* Now we're ready.  The first interrupt we will receive is the */
   /* CONNECTION COMPLETE interrupt that occurs when APPCSERV     */
   /* issues an APPC ACCEPT.                                      */
sigpause(0);

   /* Initialize the SEND parameter list with the pathid and      */
   /* buffer addresses, and the length of the reply buffer.       */
memset(&send, 0, sizeof(send));
send.pathid = pathid;
send.sendop = IPSNDRCV;
send.bf1.adr  = (char *) &logrec;
send.bf2.adr  = (char *) &reply;
send.bf2.ln  = sizeof(reply);

   /* Prompt for input messages and send until EOF.               */
fgets(logrec.message, sizeof(logrec.message), stdin);
while (!feof(stdin)) {

      /* Put message length in the SEND parameter list.           */
   logrec.length = strlen(logrec.message) + 1;
   send.bf1.ln = logrec.length;

      /* Send message.  Wait (via sigpause) for reply. Upon       */
      /* receipt of reply, 'rep_len' contains the number of       */
      /* unused bytes in the reply buffer.  Print the reply       */
      /* and prompt for a new message.                            */
   appcsdta(&send);
```

```
      sigpause(0);
      rep_len = reply.length - sizeof(reply.length);
      printf("APPCSERV replies "%.*s"\n", rep_len,
              reply.message);
      fgets(logrec.message, sizeof(logrec.message), stdin);
   }

      /* We're ready to quit.  Ask the server if it's OK to sever.    */
      /* If so, then sever.                                           */

   memset(&send, 0, sizeof(send));
   send.pathid = pathid;
   send.ip2.flags2 = IPWAIT;       /* Specify WAIT=YES.               */
   send.sendop = IPCNFSEV;

   if ((rc = appcsdta(&send)) != 2) {
      printf("Return code from appcsdta = %d\n", rc);
      if (rc == 1)
         printf("IPRCODE = x'%X'\n", send.ip1.rcode);
      exit(12);
   }

   memset(&conn, 0, sizeof(conn));
   conn.pathid = pathid;
   conn.ip2.flags2 = IPWAIT;      /* Specify WAIT=YES.                */
   conn.sendop = IPSNORM;

   if ((rc = appcsevr("APPCSEND", &conn,"ONE")) != 2) {
      printf("Return code from appcsever = %d\n",rc);
      if (rc == 1)
         printf("IPRCODE = x'%X'\n", conn.ip1.rcode);
      exit(16);
   }

      /* Terminate APPC communications for APPCSEND.                  */
   iucvclr("APPCSEND");
   exit(0);
}

#pragma eject

/* The SIGIUCV signal handler.  Signals are blocked until return.     */

void appctrap(void)
{
   appc_conn_data *XID;             /* Pointer to external interrupt  */
                                    /* data returned by siginfo.      */
   int rc;

      /* Get a pointer to the external interrupt data.  Use the       */
      /* interrupt type to determine what to do.                      */
   XID = (appc_conn_data *) siginfo();
   switch (XID->ip1.type) {
      case APPC_CONNECTION_COMPLETE:
```

```
            pathid = XID->pathid;      /* Save the pathid.              */
            break;

         case APPC_FUNCTION_COMPLETE:
                                       /* See if it's for CONFIRM.      */
            if (XID->pathid == pathid && XID->ip2.whatrc == IPSNDCNF) {
               struct appc_send_plist *send =
                  (struct appc_send_plist *)XID;
               send->sendop = IPCNFRMD;
               rc = appcscfd(send);
               if (rc != 2) {
                  printf("Error %d confirming request to recv\n", rc);
                  printf("IPRCODE = x'%X'\}n", XID->ip1.rcode);
                  exit(20);
               }
            }
            break;

         case APPC_SEVER_INTERRUPT:    /* Handle unexpected termination  */
                                       /* of APPCSERV.                   */
            puts("Unexpected SEVER!");
            exit(24);

         default:                      /* Handle unexpected signal type. */
            printf("Unexpected interrupt type x'%X'\n",XID->ip1.type);
            fflush(stdout);
            abort();
      }

         /* Re-establish this function as the SIGIUCV signal handler.    */
      signal(SIGIUCV, &appctrap);
      return;
   }
```

The APPCSERV Program

The APPCSERV program contains three functions. The **`main`** function initializes IUCV signal processing and connects to the *IDENT system service. It then waits for connection requests. As long as some program is connected, it processes interrupts. The **`appctrap`** function is the IUCV signal handler that processes incoming external interrupts. Generally these are CONNECTION PENDING interrupts, which are requests from APPCSEND to connect, and incoming message interrupts, which are messages from APPCSEND. This function can also handle the SEVER interrupts that occur if *IDENT or APPCSEND sever the path and the single CONNECTION COMPLETE interrupt that indicates the connection to *IDENT is complete. Finally, **`rcvrply`** receives an incoming message, collects a reply from the user, and sends the reply to APPCSEND.

Here is the APPSERV program:

```
#include <lcsignal.h>
#include <lcstring.h>
#include <lcio.h>
#include <cmsappc.h>
#include <stdlib.h>

void appctrap(void);                /* Declare SIGIUCV signal handler.*/
                                    /* Establish stdin prompt string. */
char *_stdiamp = "prompt=What do I reply to SENDER?\n, eof=";
int connects = 0;                   /* number of connections made     */
static short ident_pathid = 0;      /* used for *IDENT                */

/* Declare internal functions.                                        */

static void rcvrply(struct appc_send_plist *);

main()
{
   int maxconns;               /* number of IUCV connections allowed  */
   int rc;                     /* return code from functions          */

   struct iucv_path_plist conn;     /* IUCV CONNECT parameter list    */
   ident_parms   ident;             /* *IDENT parameters              */

      /* Initialize IUCV signal processing and establish a handler.   */
      /* Block IUCV signals until we're ready to handle them.         */
   signal(SIGIUCV, &appctrap);
   sigblock(1 << (SIGIUCV-1));

      /* Identify this program as APPCSERV.                           */
   if ((rc = iucvset("APPCSERV", &maxconns)) != 0) {
      printf("Return code from iucvset APPCSERV was %d\n",rc);
      exit(4);
   }
   printf("Maximum IUCV connections: %d\n",maxconns);

      /* Declare another name for connecting to *IDENT.               */
   if ((rc = iucvset("IDENTHAN", &maxconns)) != 0) {
      printf("Return code from iucvset IDENTHAN was %d\n", rc);
      exit(8);
   }

      /* Create an *IDENT parameter list.                             */
   memset(&ident, 0, sizeof(ident));
   memcpy(ident.name, "APPCSERV", 8);
   ident.fcode = MANAGE_RESOURCE;
   ident.flag  = ALLOW_SECURITY_NONE;
   ident.ntype = RESOURCE_ID;
```

```
      /* Stow "*IDENT" in IPVMID and copy *IDENT parms to CONNECT     */
      /* plist. Execute an IUCV CONNECT to *IDENT.                    */
   memset(&conn, 0, sizeof(conn));
   memcpy(conn.vmid, "*IDENT  ", 8);
   memcpy(conn.pgm, &ident, sizeof(ident));

   rc = iucvconn("IDENTHAN", &conn);
   if (rc != 0) {
      printf("Return code from iucvconn IDENTHAN was %d\n", rc);
      if (rc == 1)
         printf("IPRCODE = x'%X'\n",conn.ip.rcode);
      exit(12);
   }
   ident_pathid = conn.pathid;      /* Save IDENTHAN pathid.          */
   sigpause(0);                     /* Wait for CONNECTION COMPLETE   */
                                    /* from *IDENT.                   */

      /* Wait for initial CONNECTION PENDING from APPCSEND.           */
   sigpause(0);

      /* As long as some SENDER is connected, wait for incoming       */
      /* messages.                                                    */
   while (connects > 0) sigpause(0);

      /* All paths have been terminated.  Terminate IUCV processing.  */
   iucvclr("APPCSERV");
   iucvclr("IDENTHAN");
   exit(0);
}

#pragma eject

/* SIGIUCV signal handler.  Signals are blocked until return.         */

void appctrap(void)
{
   struct appc_send_plist recv;     /* used to receive allocate       */
   struct appc_send_plist *send;    /* APPC SEVER plist pointer       */
   appc_conn_data *XID;             /* Pointer to external interrupt  */
                                    /* data returned by siginfo.      */
   int rc;                          /* return code from iucvacc       */
   ident_parms ident;               /* so we can test RCODE           */
   char  *allocate_data;            /* ptr to allocate data           */

      /* Get a pointer to the external interrupt data.  Use the       */
      /* interrupt type to determine what to do.                      */
   XID = (appc_conn_data *) siginfo();
   switch (XID->ip1.type) {
      case APPC_CONNECTION_PENDING: /* Retrieve allocation data.      */
         allocate_data = malloc(XID->bf2.ln);
         memset(&recv, 0, sizeof(recv));
         recv.pathid = XID->pathid;
         recv.ip2.flags2 = IPWAIT;
         recv.bf1.adr = allocate_data;
```

```
   recv.bf1.ln = XID->bf2.ln;
   rc = appcrecv(&recv);
   if (rc != 2) {
      printf("Error %d receiving allocation data\n", rc);
      printf("IPRCODE = x'%X'\n", recv.ip1.rcode);
      printf("IPAUDIT = %x\n", recv.audit);
      exit(16);
   }
   XID->ip1.type = 0;         /* Accept the connection          */
   XID->ip2.flags2 = 0;
   if ((rc = iucvacc("APPCSERV", (struct iucv_path_plist *)
        XID)) != 2) {
      printf("Return code from iucvacc = %d\n", rc);
      if (rc == 1)
         printf("IPRCODE = x'%X'\n", XID->ip1.rcode);
      exit(20);
   }
   connects++;                /* Keep track of the number of    */
   break;                     /* connections made.              */
case APPC_INCOMING_MSG:       /* Call message handler.          */
   rcvrply((struct appc_send_plist *)XID);
   break;
case PATH_SEVERED:            /* *IDENT decided to stop.        */
   if (XID->pathid == ident_pathid) {
      memcpy(&ident, XID->resid, sizeof(ident));
      if (ident.rcode != 0x00) {
         printf("*IDENT path severed.rcode = %x\n",
                 ident.rcode);
         exit(24);
      }
   }
   else {                     /* An IUCV sever?  Really?        */
      printf("Unexpected sever.IPPATHID = %d\n", XID->pathid);
      exit(28);
   }
   break;
case CONNECTION_COMPLETE:    /* *IDENT connection complete     */
   if (XID->pathid == ident_pathid)
      puts("Connection to *IDENT complete\n");
   else {                     /* Some other connection?         */
      printf("Unexpected connection complete.Path = %d\n",
              XID->pathid);
      exit(32);
   }
   break;
case APPC_SEVER_INTERRUPT:    /* APPCSEND decided to stop.      */
   send = (struct appc_send_plist *) XID;
   send->ip2.flags2 = IPWAIT;
   send->sendop = IPSNORM;
   send->bf1.adr = NULL;
   send->bf1.ln = 0;
```

```
         if ((rc = appcsevr("APPCSERV", send, "ONE")) != 2) {
            printf("Return code from appcsevr = %d\n", rc);
            printf("IPRCODE is x'%X'\n", send->ip1.rcode);
            exit(36);
         }
         connects--;                /* Update number of connections.  */
         break;
      default:                      /* Handle other interrupt types.  */
         printf("Unexpected interrupt type %d\n",
            XID->ip1.type);
         fflush(stdout);
         abort();
   }

      /* Reestablish this function as the SIGIUCV signal handler.     */
   signal(SIGIUCV, appctrap);
   return;
}

#pragma eject

/* function to issue APPC RECEIVE and print the message               */

static void rcvrply(struct appc_send_plist *int_data)
{
   int msg_len;                     /* incoming message length        */
   struct senddata {
      unsigned short length;        /* length of data to send         */
      char message[120+2];          /* message buffer                 */
   } logrec;
   int rc = 0;                      /* return code                    */

      /* Create APPC RECEIVE parameter list using the external        */
      /* interrupt data area.  Issue the APPC RECEIVE.                */
   int_data->ip2.flags2 = IPWAIT;
   int_data->bf1.adr = (char *) &logrec;
   int_data->bf1.ln = sizeof(logrec);
   rc = appcrecv(int_data);
   if (rc != 2) {
      printf("Return code from receive = %d\n", rc);
      printf("IPRCODE is x'%X'\n", int_data->ip1.rcode);
      exit(40);
   }

      /* Perhaps the sender wants to sever.  If so, confirm sever.    */
   if (int_data->ip2.whatrc == IPCNFSEV) {
      int_data->sendop = IPCNFRMD;
      if ((rc = appcscfd(int_data)) != 2) {
         printf("Error %d attempting to confirm sever\n", rc);
         printf("IPRCODE is x'%X'\n", int_data->ip1.rcode);
         exit(44);
      }
      return;
   }
```

```
      /* If the return code is anything else, bad news.              */
   if (int_data->ip2.whatrc != IPSEND) {
      puts("Protocol error.  Sender not in receive state\n");
      exit(48);
   }

      /* Upon return, int_data->bf2.ln contains the number of unused  */
      /* bytes in the message buffer.  Print the message.             */
   msg_len = sizeof(logrec) - int_data->bf2.ln - 2;
   printf("SENDER says \"/%.*s\"\n", msg_len, logrec.message);
      /* Prompt for a reply message.  If EOF, quit.                   */
   fgets(logrec.message, sizeof(logrec.message), stdin);
   if (feof(stdin)) {
      puts("Terminating due to end-of-file.\n");
      exit(52);
   }

      /* Fill in APPC SENDDATA parameter list with buffer             */
      /* address/length send reply.                                   */
   int_data->ip2.flags2 = IPWAIT;
   int_data->sendop = IPDATA;
   int_data->bf1.adr = (char *) &logrec;
   logrec.length = strlen(logrec.message) + 1;
   int_data->bf1.ln = logrec.length;
   rc = appcsdta(int_data);
   if (rc != 2) {
      printf("Error %d on reply SENDDATA\n",
      printf("IPRCODE is x'%X'\n", int_data->ip1.rcode);
      exit(56);
   }

      /* We are still in SEND state. Ask to turn conversation around. */
   int_data->ip2.flags2 = IPWAIT;
   int_data->sendop = IPPREPRC;
   if ((rc = appcscnf(int_data)) != 2) {
      printf("Error requesting confirmation to receive. rc =%d", rc);
      printf("IPRCODE = x'%X'\n", int_data->ip1.rcode);
      exit(60);
   }
}
```

Guidelines and Cautions

For additional information concerning APPC/VM support, see the IBM documentation.

When using SAS/C APPC/VM functions, you should be aware that if the value of the **code** argument in the **iucvsevr** function is not **''ALL''**, **''ONE''** is assumed.

For more information on handling program interrupts and communications, refer to Chapter 5, "Signal-Handling Functions" in *SAS/C Library Reference, Volume 1.*

7 The Subcommand Interface to EXECs and CLISTs

7-1 Introduction
7-1 Subcommand Processing in C Programs
7-2 Steps in Subcommand Processing
7-3 Overview of the SUBCOM Environment
7-3 SUBCOM and CMS
7-3 SUBCOM and TSO
7-4 SUBCOM and OpenEdition
7-5 SUBCOM in Interactive and Noninteractive Environments
7-5 Function Descriptions
7-27 Examples of SUBCOM Processing
7-29 CLIST Example for SUBCOM Processing
7-30 TSO REXX Example for SUBCOM Processing
7-31 CMS EXEC Example for SUBCOM Processing
7-32 Guidelines for Subcommand Processing

Introduction

The SUBCOM feature provides an interface between the SAS/C library and the command languages of CMS and TSO (REXX or EXEC2 and CLIST). It can also be used as an interface to REXX for programs running under the OpenEdition shell. The interface allows a program to accept input (traditionally subcommands) from a running EXEC or CLIST or to begin executing a new EXEC or CLIST. The SUBCOM interface is implemented through a set of nine functions. This discussion of the SUBCOM feature provides an introduction and overview of the SUBCOM interface followed by descriptions and examples of each of the functions involved.

This chapter is intended for systems programmers and assumes an understanding of either the CMS or TSO command language as well as the concepts of EXEC processing.

Subcommand Processing in C Programs

Subcommand functions enable a program to identify itself to the operating system as a subcommand processor. The ability to handle subcommands provides program flexibility and extends program capabilities. The SUBCOM facility provides a program with a mechanism for taking input from CMS or TSO command languages and provides the programmer with an additional full-scale language to perform program tasks. For example, a simple subcommand application allows a CLIST or EXEC to execute a number of similar commands repeatedly. A more complicated application defines complex subcommands as sequences of simpler commands, packaged as a CLIST or EXEC.

All SUBCOM interfaces are portable between CMS, TSO, and OpenEdition, so a program written for one environment can be ported to another. Although the exact internal processing (or effect of SUBCOM functions) is system-dependent, these differences are transparent to the user, so a program's operation on different systems is still predictable. Even SUBCOM programs with substantial differences in required CMS and TSO behavior can usually be written with the system dependencies isolated to smaller parts of the program.

Steps in Subcommand Processing

Under TSO, CMS and the OpenEdition shell, programs that support a subcommand environment can be described as doing their processing in a basic loop consisting of the following steps:

1. Get the next subcommand (from the terminal or from an EXEC or CLIST).
2. Identify the subcommand name.
3. Process the subcommand, possibly issuing one or more messages.
4. Set the subcommand's return code for access by an EXEC or CLIST.
5. Go to step 1.

Associated with each of these steps are the following SUBCOM functions:

1. **`execget`**, which obtains a subcommand from an appropriate source
2. **`execid`**, which extracts the subcommand name and may perform additional system-dependent processing, for example, recognizing system-defined use of special characters in the subcommand
3. **`execmsg`**, which sends diagnostic messages, the exact form of which can be controlled by the user
4. **`execrc`**, which sets the return code from the completed subcommand and may perform additional system-dependent processing.

In addition, an **`execinit`** function is used to initialize the subcommand environment (and assign a name to it), and **`execend`** is used to terminate the environment.

The set of library functions that implement the SUBCOM interface can be summarized as follows: **`execinit`** establishes and assigns a name to a SUBCOM environment.

`execcall` invokes a new CLIST or EXEC, from which input can be read later.

`execend` cancels the program's SUBCOM environment.

`execget` gets a line of input from an EXEC/CLIST or from the terminal.

`execid` parses a string into a subcommand name and operands, using system-dependent conventions.

`execinit` establishes and assigns a name to a SUBCOM environment.

`execmsg` sends a message to the user, in a system-dependent way, allowing the use of system facilities for message suppression or abbreviation.

`execmsi` tests the environment, in a system-dependent way, to determine which portions of messages the user wants to print.

`execrc` sets the return code of the most recent subcommand.

`execshv` provides a way to fetch, set, or drop REXX, EXEC2 or CLIST variable values.

If these functions are used to create a subcommand environment, a typical SUBCOM program has the following basic form:

```
execinit(...);                  /* Set up environment.       */
for(;;) {
 execget(...);                  /* Get a subcommand.         */
    cmd = execid(...);          /* Identify subcommand.      */
```

```
    if (strcmp(cmd,"END")==0)
       break;
    if (strcmp(cmd,"EXEC")==0)
       execcall(...);            /* Implement exec.          */
    else
       process(...);             /* Process the subcommand.  */

    execrc(...);                 /* Set the return code.     */
}
execend();                       /* Terminate the environment.*/
```

Overview of the SUBCOM Environment

As noted in the **Introduction**, a program using the SUBCOM interface can be completely portable between CMS, TSO and OpenEdition environments. Topics of general interest concerning these implementations are discussed in this section. Using SUBCOM in an interactive and noninteractive environment also is discussed. (Topics specific to the implementation of each SAS/C function are presented later in each detailed function description.)

SUBCOM and CMS

Under CMS, you traditionally establish a subcommand environment by invoking the CMS SUBCOM service. You then write an assembler routine to provide the appropriate function parameters and calling sequence. The SAS/C compiler simplifies this process by providing a set of functions that enable a program to become a subcommand processor. More detailed coverage of all aspects of the CMS SUBCOM service can be found in the appropriate IBM documentation for your system.

To use the SUBCOM facility under CMS, a program must tell CMS its name and the address to which CMS should pass the commands. The next step is to give the program the commands. Under CMS the user typically uses an EXEC to pass the commands. Because commands in an EXEC are sent by default to CMS, however, an alternate destination can be specified via the address statement (in REXX) or the &SUBCOMMAND statement (in EXEC2). After the program has handled the command, it returns to CMS so that CMS can get another command for the program to handle. Therefore, subcommand processors are being invoked continually and returning to CMS until they are told to stop or are interrupted.

CMS also has a facility for letting the user program call the EXEC processor directly. In this case, the EXEC processor sends commands by default to the program instead of CMS. To invoke an EXEC for itself, the program again must first declare itself via SUBCOM so that it can get commands. In this case, it then calls CMS to tell it the name of a file that the EXEC processor will execute. The EXEC processor executes the file and sends the commands to the program.

To facilitate this processing, the SUBCOM interface uses the CMS SUBCOM facility that enables a subcommand entry point. To invoke an EXEC, the interface calls the EXEC processor with the filename and filetype of the EXEC to be executed.

SUBCOM and TSO

Under TSO, CLISTs are executed through a two-stage process. First, the EXEC command is called to read and compile the CLIST. The resulting data are accessible through a control block called the *TSO stack*. Later, when a program, or TSO itself, needs to obtain input and is willing to accept CLIST input, it calls the PUTGET service routine. If a CLIST is present on the stack, TSO passes control to the CLIST. The CLIST executes until it generates a subcommand line. At this point, TSO returns control to PUTGET, which passes the subcommand line back to its caller.

The SUBCOM implementation for TSO is closely tied to traditional CLIST processing. The **`execget`** function reads the CLIST input by calling the PUTGET

routine. The **execcall** function, which invokes a new CLIST, does so by calling the EXEC command.

Accessing REXX under TSO

The SAS/C Library SUBCOM features supports TSO REXX in addition to the CLIST language. The REXX support does not require that existing CLIST applications be modified. A single application can generally be used to communicate with both CLISTs and REXX EXECs without concern for which command language is used. There are a few cases where a program's behavior processing a REXX EXEC may differ from the behavior processing a similar CLIST, as described in the function descriptions.

When using the SUBCOM interface to communicate with a REXX EXEC in TSO, the EXEC must specifically address (via the ADDRESS command) the SAS/C application's environment to send it subcommands. The default environment for EXECs invoked from SAS/C is still TSO. The name to be addressed is the value of the first argument to **execinit**. Refer to ** UNRESOLVED HEAD REFERENCE REFID=rexxex ** for additional information.

Note that a REXX EXEC that addresses a SUBCOM application cannot be called using the **system** function. If this is attempted, any subcommand addressed to the SUBCOM application will fail with a return code of **-10**.

The TSO SUBCOM implementation for calling a REXX EXEC is somewhat different from the CLIST implementation. In this case, **execinit** defines a host-command environment routine to process subcommands addressed to the application. When **execget** is called, control is transferred from the SUBCOM application to REXX. When the REXX EXEC generates a subcommand for the application, REXX calls the Host Command Environment routine, which passes the subcommand back to **execget**.

Attention Handling

If a SUBCOM application has a SIGINT signal handler or uses the debugger, an attention while a REXX EXEC is running is specially handled. The user is prompted to enter either IC or a regular REXX immediate command. If IC is entered, an attention is presented to the debugger or the SIGINT handler as if REXX were not active. Otherwise, the REXX immediate command is passed to REXX. This allows attention handling to be shared between REXX (which ordinarily will not let an application handle attentions itself) and the SAS/C application.

SUBCOM and OpenEdition

Under the OpenEdition shell, REXX EXECs are stored as executable files in the hierarchical file system, and can be invoked by any of the **exec** family of functions. However, when a REXX EXEC is invoked in this fashion, it runs in a separate address space from its caller, and only the standard REXX environments (such as the MVS and SYSCALL environments) are available.

The SAS/C SUBCOM implementation for OpenEdition uses the **oeattach** function to invoke REXX in the same address space as its caller. This allows the library to establish the additional SUBCOM environment used for communication with the calling program. The IBM support routine BPXWRBLD is used to establish a REXX environment that is OpenEdition aware, so that REXX input and output will be directed to the hierarchical file system rather than to DDnames.

The SAS/C REXX interface under the shell runs as a separate process from the calling program to prevent interference between the program and EXEC processing. When the REXX interface is created by a call to **execinit**, the program's environment variables and file descriptors are copied to the REXX interface process. This copy of the environment is not affected by any changes requested by the program, including the use of **setenv** or **close**. REXX input and output are always

directed to the REXX process' file descriptors 0 and 1, even if these descriptors have been reopened by the program after the call to **execinit**.

Note that a REXX script invoked by the **system** function under OpenEdition cannot address the invoker's SUBCOM environment. Any attempt to do so will fail with return code **-3** (command not found).

SUBCOM in Interactive and Noninteractive Environments

A program operating in a subcommand environment can execute in either an interactive or noninteractive manner. Either type of execution can be used by programs that accept input from CLISTs or EXECs, subject to operating system restrictions.

A program with interactive style gets subcommands from the terminal. This style program, even if it is invoked from an EXEC or CLIST, begins by reading from the terminal and accepts EXEC or CLIST input only as the result of terminal input (such as an EXEC command from the user). Commands in an EXEC or CLIST following a call to an interactive program can be executed only after that program terminates.

A program with noninteractive style receives subcommands from an EXEC or CLIST. If this style program is invoked from an EXEC or CLIST, it gets its input from that EXEC or CLIST and takes input from the terminal only as a result of commands from the EXEC or CLIST or after that EXEC or CLIST has terminated.

For example, an EXEC or CLIST could contain the following commands:

```
CPGM1   (options)
SUBCMD2 (options)
```

If the program CPGM1 is interactive, the SUBCMD2 input line is not read by CPGM1 or processed at all until CPGM1 terminates. If CPGM1 is noninteractive, however, the SUBCMD2 line is read by CPGM1 the first time it calls **execget** to get a line of input.

Under CMS, most processors are defined as interactive. In versions of CMS after VM/SP 5, we do not recommend noninteractive programs.

Under TSO, most processors are defined as noninteractive. An interactive program must create a new version of the TSO stack to avoid reading input from the CLIST that invoked it.

Under the OpenEdition shell, only the interactive mode of SUBCOM is supported.

Function Descriptions

Descriptions of each subcommand interface function follow. 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. A comprehensive example incorporating all the SUBCOM functions is presented and discussed following the function descriptions. This chapter concludes with a summary of guidelines and considerations for using the SUBCOM interface.

execcall Identify a Macro to Be Executed

SYNOPSIS

```
#include <exec.h>

int execcall(const char *cmd);
```

DESCRIPTION

The **execcall** function invokes an EXEC or CLIST. The character string pointed to by **cmd** is an EXEC command. The exact format of the command is system-dependent. **cmd** can include a specification of the file to execute and also operands or options.

The library assumes that both **''EXEC cmdname operands''** and **''cmdname operands''** are valid arguments to **execcall**, but the exact details are system-dependent. (Under OpenEdition, the command string should not begin with **''EXEC''** unless the intent is to invoke a REXX script named **EXEC**.)

RETURN VALUE

The **execcall** function returns **0** if the call is successful or nonzero if the call is not successful. A nonzero return code indicates that the EXEC or CLIST could not be executed.

Under TSO, the return code is that of the EXEC command or a negative return code whose meaning is the same as that from the **system** function. However, if **execcall** is used to call a TSO REXX EXEC that terminates without passing any subcommands to the C program, the return code is that set by the EXEC's RETURN or EXIT statement. This cannot always be distinguished from an EXEC command failure.

Under the OpenEdition shell, as under TSO, the return code from **execcall** may reflect either a failure of the REXX interface (IRXJCL) or a return code set by the called EXEC.

CAUTIONS

The **execcall** function is valid only for programs that have defined an environment name through **execinit**.

Once **execcall** is used successfully, future **execget** calls are expected to read from the EXEC or CLIST. **execget** returns **"*ENDEXEC** *rc***"** the first time it is called after a CLIST or EXEC invoked by **execcall** has completed, where *rc* is the return code from the CLIST or EXEC. Subsequent calls to **execget** result in terminal reads.

IMPLEMENTATION

For a CMS EXEC or a TSO CLIST, **execcall** does not cause any of the statements of the EXEC or CLIST to be executed. **execcall** finds the file to be executed and completes the parameter list to be processed.

Under CMS, the macro to be invoked has a filename of **execname** and a filetype of **envname**, where **envname** was specified by the call to **execinit**. If the macro cannot be found, an error code is returned. Otherwise, **execcall** saves the fileid but does not invoke the macro. The next call to **execget** invokes the EXEC processor with the EXEC parameter list and a FILEBLOK indicating the filename (from **execcall**) and filetype (from **execinit**) of the macro.

execcall Identify a Macro to Be Executed

(continued)

Under TSO, **execcall** uses the ATTACH macro to call the EXEC command. Any CLIST arguments are processed at this time, and prompts are generated for any omitted arguments. No other CLIST processing occurs until **execget** is called. For a TSO REXX EXEC, **execcall** executes the statements of the EXEC up to the first subcommand addressed to the program's environment before control is returned to the program.

Note that if errors are detected by the EXEC command, the TSO stack is flushed, and a diagnostic message is printed by EXEC.

Under the OpenEdition Shell, the requested REXX EXEC is invoked using the IRXJCL service routine. Due to limitations of this service, the name of the EXEC is limited to eight characters. The EXEC must be an executable file, and must reside in one of the directories defined by the **PATH** environment variable. The statements of the EXEC up to the first subcommand addressed to the program's environment will be processed before control is returned to the program.

EXAMPLE

See the example for the **execget** function and the comprehensive SUBCOM example.

execend Cancel the SUBCOM Interface

SYNOPSIS

```
#include <exec.h>

int execend(void);
```

DESCRIPTION

The **execend** function is called to terminate subcommand processing.

RETURN VALUE

The **execend** function returns `0` if the call is successful or nonzero if the call is not successful. A nonzero value is returned when there is no previous successful call to **execinit** or if **execend** encounters other errors during its processing. Further use of a SUBCOM environment created by **execinit** is not possible after a call to **execend**, if **execend** does not complete successfully.

CAUTIONS

If any EXECs invoked by **execcall** are active when **execend** is called, the effect is system-dependent and also can depend on whether the previous call to **execinit** was interactive.

Under CMS, if the program invokes an EXEC via **execcall** and then calls **execend** (see the IMPLEMENTATION section) before processing the entire EXEC, **execend** allows the EXEC processing to complete. All remaining subcommands are accepted but ignored, and the return code for each subcommand is set to `-10`.

Under TSO, if the original **execinit** call is interactive, any active CLISTs started by **execcall** are terminated when **execend** is called. If the **execinit** call is noninteractive, however, any such CLISTs continue to execute and may generate input for the processor that called the C program.

If the **execend** function is called with one or more TSO REXX EXECs still active, the EXECs continue to execute, but any attempt to send a subcommand to the application that called **execend** results in a REXX RC of `-10` and a message. The call to **execend** is not considered to be complete until all active REXX EXECs have terminated. If **execinit** is called in noninteractive mode, any CLISTs on the TSO stack beneath the first REXX EXEC at the time of a call to **execend** remain active.

Under the OpenEdition shell, the behavior is the same as under CMS.

IMPLEMENTATION

Under TSO, when SUBCOM is initialized with an interactive call to **execinit**, **execend** causes all CLISTs invoked by **execcall** to be forcibly terminated. If **execinit** creates a new stack allocation, the previous allocation is restored.

Similarly, if SUBCOM is initialized as interactive, **execend** deletes the current REXX data stack, restoring the stack defined when **execinit** was called. If SUBCOM is initialized as noninteractive, the REXX data stack is not modified.

execend Cancel the SUBCOM Interface
(continued)

EXAMPLE

See the example for the **`execget`** function and the comprehensive SUBCOM example.

execget Return the Next Subcommand

SYNOPSIS

```
#include <exec.h>

char *execget(void);
```

DESCRIPTION

The **execget** function obtains a subcommand from a CLIST or EXEC or from the terminal if no CLIST or EXEC input is available. It returns the address of a null-terminated string containing the subcommand. The area addressed by the pointer returned from **execget** remains valid until the next call to **execget**.

If an EXEC or CLIST is active and accessible, input is obtained from that EXEC or CLIST; otherwise, the user is prompted with the name of the environment and a colon, and the next line of terminal input is returned to the program. (Before the prompt is issued, input is taken from the REXX data stack, if the stack is not empty.) If the SUBCOM interface is initialized with an interactive call to **execinit**, any EXECs or CLISTs active at the time are inaccessible. Additionally, under TSO, any input on the REXX data stack is inaccessible. Therefore, for such programs, a call to **execget** reads from the terminal unless **execid** or **execcall** has invoked a new CLIST or EXEC.

If all active EXECs or CLISTs terminate and the data stack becomes empty, **execget** returns a pointer to the string **"*ENDEXEC** *rc*". The component **''*ENDEXEC''** is a literal character string provided as a dummy subcommand. The component *"rc"* is an appropriate return code. The *"rc"* can be omitted if no return code is defined. Subsequent calls to **execget** issue terminal reads, using a prompt formed from the original **envname** suffixed with a colon.

The **''*ENDEXEC''** convention was designed for the convenience of programs that want to use **execget** to read only from CLISTs or EXECs, while using another form of normal input (for instance, a full-screen interface). Such programs use **execget** only after **execcall** (or at start-up, if noninteractive) and switch to their alternate form of input after **execget** returns **''*ENDEXEC''**.

RETURN VALUE

The **execget** function returns a pointer to a buffer containing the subcommand string. If no subcommand can be returned due to an unexpected system condition, **execget** returns **NULL**.

CAUTIONS

The **execget** function is rejected when there is no environment name defined via **execinit**.

Under TSO, a CLIST can use the TERMIN statement to switch control from the CLIST to the terminal. Use of TERMIN when the SUBCOM interface is initiated with a noninteractive call causes **''*ENDEXEC''** to be returned, even when the CLIST has not ended.

The CMS SUBCOM interface acknowledges two logical cases of SUBCOM processing. The first case is when the program is accepting subcommands from an EXEC that is invoked external to the program. The second case occurs when the program is accepting subcommands from an EXEC invoked by **execcall** (see the **execcall** description).

execget Return the Next Subcommand
(continued)

If the program is accepting subcommands from an EXEC invoked externally and the EXEC terminates without sending a subcommand to **execget**, the string **''*ENDEXEC''** is placed in the subcommand buffer without a return code value because the return code belongs to the environment that invoked the EXEC (usually CMS). Subsequent calls to **execget** read from the terminal. If the program terminates before all the subcommands in the EXEC are processed, the remaining subcommands are issued to CMS. Under most circumstances, CMS responds with an error message and a return code of **-3**. (Refer also to the note on ***ENDEXEC** in the section CAUTIONS under **execcall**.)

Note that under the OpenEdition shell, input is read from the REXX process' file descriptor 0 if no EXEC is active and there is no data present on the data stack. File descriptor 0 is normally allocated to the terminal, but could be directed elsewhere if the program's standard input had been redirected when **execinit** was called.

Premature termination of programs that have invoked an EXEC via **execcall** are handled by **execend**. See the section CAUTIONS following the **execend** description.

IMPLEMENTATION

For CMS, refer to the IMPLEMENTATION discussion in the **execcall** description.

Under TSO, **execget** issues the PUTGET MODE macro to get a line of input if the current input source is a CLIST or the terminal. If the current input source is a REXX EXEC, the next input line is obtained from the host-command environment routine. In interactive mode, **''*EXECEND''** is returned when the current allocation of the TSO stack becomes empty. In noninteractive mode, **''*EXECEND''** is returned when control passes from a CLIST to the terminal, or on the first call if no CLIST is active when program execution begins.

Under OpenEdition, if no EXEC is active when **execget** is called, **execget** reads a line from the REXX process' file descriptor 0 using the IRXSTK service routine's PULLEXTR function.

execget Return the Next Subcommand
(continued)

EXAMPLE

This example obtains a subcommand from a CLIST or EXEC. If no command is available, SUBCOM processing is terminated. Otherwise, the command is assumed to be an EXEC command and is executed using **execcall**.

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

int rc;
char *cmd;

cmd = execget();

if (cmd != 0 && memcmp(cmd,"*ENDEXEC",8))
   execend();
else {
   rc = execcall(cmd);
   execrc(rc);
}
```

Note: Also see the comprehensive SUBCOM example.

execid Parse a Line of Input as a Subcommand

SYNOPSIS

```
#include <exec.h>

char *execid(char **cmdbuf);
```

DESCRIPTION

The **execid** function parses a line of input as a subcommand, according to system-specific conventions. **cmdbuf** is a pointer to the address of the line of input. When **execid** returns, ***cmdbuf** is altered to address the first operand of the subcommand or the null character ending the string. The return value from **execid** is a pointer to the subcommand name.

The **execid** function can enforce syntax conventions of the host system. This is useful particularly in TSO applications. For example, under TSO, **execid** checks to see if the subcommand begins with the **'%'** character, and then treats it as an implicit EXEC command if it does. In such cases, **execid** can preempt processing of the subcommand. If it does, it returns a command name of **``*EXEC''** to indicate that processing of the subcommand has been preempted, that a new EXEC may have been invoked, and that the program should get another line of input.

If the PCF-II product is installed on TSO, **execid** also processes the PCF-II X command and returns **``*EXEC''** on completion of PCF-II processing.

Under MVS and CMS, **execid** uppercases the input command name in accordance with TSO and CMS conventions. Under the OpenEdition shell, the command name is not uppercased, in accordance with UNIX conventions.

RETURN VALUE

The **execid** function returns a pointer to the subcommand name if the call is successful or **0** if it failed.

A **0** return code from **execid** always means that the program should expect subsequent input to come from a CLIST or an EXEC. (The next **execget** that reads from the terminal always returns **``*ENDEXEC''**.)

CAUTIONS

Whether **execid** is valid for a program without an environment name defined is system-dependent.

The **execid** function does no validation of the subcommand name. It is the responsibility of the program to decide whether to accept the subcommand.

Under TSO, if **execid** is not called for a subcommand, the CLIST variable &SYSSCMD does not contain the current subcommand name.

IMPLEMENTATION

The use of **execid** is optional. However, the environmental integration it provides may be helpful in TSO applications.

Under TSO, **execid** calls the IKJSCAN service routine to extract the subcommand name. If IKJSCAN indicates that the % prefix was used, **execid** calls the EXEC command to process an implicit CLIST request and returns **``*EXEC''** to indicate this.

execid Parse a Line of Input as a Subcommand
(continued)

EXAMPLE

```
#include <exec.h>

char *cmdname, *operands;
cmdname = execid(&operands);
```

Note: Also see the comprehensive SUBCOM example.

execinit Create a SUBCOM Environment

SYNOPSIS

```
#include <exec.h>

int execinit(const char *envname, int interactive);
```

DESCRIPTION

The **execinit** function establishes a SUBCOM environment with the name specified by **envname**. The characteristics of **envname** are system-dependent. Under TSO, the value of **envname** is available through the standard CLIST variable &SYSPCMD or the REXX call SYSVAR(SYSPCMD). Under CMS, the value of **envname** is the environment addressed by an EXEC to send subcommands. Also under CMS, for EXECs invoked via **execcall**, the environment is used as the filetype of the EXEC.

The **interactive** argument is a true-or-false value (where `0` is false and nonzero is true) indicating whether the program should run as an interactive or noninteractive SUBCOM processor. Because the **interactive** argument to **execinit** can be a variable, it is possible for a program to choose its mode according to a command-line option, allowing complete flexibility. An interactive processor is one that normally accepts input from the terminal, even if the program is invoked from an EXEC or CLIST. A noninteractive processor is one that is normally called from an EXEC or CLIST and that takes input from the EXEC or CLIST that calls it. Using this terminology, most CMS processors, such as XEDIT, are interactive, while most TSO processors, such as TEST, are noninteractive.

Note that TSO REXX does not allow a program invoked from an EXEC to read input from the same EXEC. For TSO REXX applications, the only distinction between interactive and noninteractive mode is that, in interactive mode a new REXX data stack is created, while in noninteractive mode the previous data stack is shared with the application.

Similarly, noninteractive mode is not supported under the OpenEdition shell.

RETURN VALUE

The **execinit** function returns `0` if the call is successful or nonzero if the call is not successful. Reasons for failure are system-dependent. Possible errors include a missing or invalid **envname** or, under CMS, a nonzero return code from the CMS SUBCOM service.

CAUTIONS

A program cannot call **execinit** more than once without an intervening call to **execend**. Under CMS, a program that calls **execinit** must have been called directly from CMS. Calls from other languages, including assembler front ends, are not supported.

Noninteractive SUBCOM processing is not supported under bimodal CMS or OpenEdition.

Under TSO, when a program calls **execinit** with **interactive** set to true (nonzero), it is not possible for CLISTs invoked by that program to share GLOBAL variables with the CLIST that called the program.

execinit Create a SUBCOM Environment
(continued)

If the value of **interactive** is true (nonzero) and the SAS/C program is invoked from an EXEC or CLIST, the remaining statements of the EXEC or CLIST are inaccessible via **execget**. If **execget** is called before the use of **execcall**, or after any EXECs or CLISTs called by **execcall** have terminated, the program gets input from the terminal. If the value of **interactive** is false (**0**) and the SAS/C program is invoked from a CMS EXEC or TSO CLIST, then calls to **execget** without the use of **execcall** read from that EXEC or CLIST.

Under TSO, if the value of **interactive** is true, a new allocation of the REXX data stack is created, and the previous data stack contents are inaccessible until **execend** is called.

IMPLEMENTATION

Under CMS, **execinit** invokes the CMS SUBCOM service. The name of the environment is **envname**, truncated to eight characters if necessary and translated to uppercase.

Under CMS, the **execinit** function creates a nucleus extension called L$CEXEC with the ENDCMD attribute. This is done so that premature EXEC termination can be detected (see the **execget** function description). If the CMS command NUCXDROP L$CEXEC is issued to delete this nucleus extension, the program continues to execute, but premature termination is not detected. If this happens, the program remains active, but control cannot be transferred from CMS back to the program. Therefore, it is not recommended that you use NUCXDROP L$CEXEC. For more information, see "Guidelines for Subcommand Processing" at the end of this chapter.

Under TSO, the value of **envname** is stored in the ECTPCMD field of the Environment Control Table (ECT). When the value of **interactive** is true, a new allocation of the TSO stack is created, if necessary, to preserve the status of a previously executing CLIST.

Under the OpenEdition shell, **execinit** creates a new process using the **oeattach** function. This process invokes the BPXWRBLD service to create an OpenEdition REXX language environment.

EXAMPLE

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

int rc;
.
.
.
if (rc = execinit("EXENV",0))
   printf("Failed to establish SUBCOM envr\n");
```

Note: See also the comprehensive SUBCOM example.

execmsg Send a Message to the Terminal

SYNOPSIS

```
#include <exec.h>

int execmsg(const char *id, char *msg);
```

DESCRIPTION

The **execmsg** function sends a message to the terminal, editing it in a system-dependent way. The character string pointed to by **id** is a message identifier that either can be sent or suppressed. **id** can be **0** to indicate the absence of the message identifier. The character string pointed to by **msg** is the actual message text.

RETURN VALUE

The **execmsg** function returns **0** if the call is successful or nonzero if the call fails. Reasons for failure are system-dependent. For example, under TSO a message cannot be sent if it is longer than 256 characters, including the identifier. **execmsg** cannot fail under CMS, although depending on the current EMSG setting, the message may not be sent.

CAUTIONS

Under CMS, the message ID must be ten characters long. If it is longer than ten characters, it is truncated on the right. If it is less than ten characters, it is padded on the left with asterisks (*). This means that a message ID of length 0 has different effects than if **id** is **NULL**, although the TSO effects are the same.

Under CMS, the maximum message length, for the ID and text, is 130 characters. If it is longer, the message text is truncated.

The **execmsg** function edits the message according to the user's EMSG setting. No message is sent in the following situations:

- if EMSG is OFF
- if EMSG is CODE and **id** is **NULL**
- if EMSG is TEXT and **msg** is **NULL**.

Under TSO, messages sent using **execmsg** can be trapped by the CLIST command output trapping facility (the symbolic variables &SYSOUTTRAP and &SYSOUTLINEnnnn). Terminal output sent in other ways (such as using the standard SAS/C I/O facilities) is not trapped.

Under OpenEdition, messages sent using **execmsg** are sent to file descriptor 1 of the REXX interface process, which is normally the terminal. OpenEdition does not support suppression of message IDs.

IMPLEMENTATION

The message is sent under TSO using PUTLINE INFOR (or PUTLINE DATA if **id** is **0**). Under CMS, the message is edited via DIAG X'5C' and sent to the terminal by TYPLIN or LINEWRT. Under OpenEdition, the message is sent to the REXX process' file descriptor 1 using the IRXSAY service routine.

The **execmsg** function can be used by programs that have not defined an environment name with **execinit** under CMS or TSO; however, prior use of **execinit** is required under OpenEdition.

execmsg Send a Message to the Terminal
(continued)

EXAMPLE

```
#include <exec.h>

static char *id="EXEC-MSG";
execmsg(id,"File not found");
```

Note: See also the comprehensive SUBCOM example.

execmsi Return System Message Preference

SYNOPSIS

```
#include <exec.h>

int execmsi(void);
```

DESCRIPTION

The **execmsi** function tests the user's preferences for printing system messages, specifically, whether message IDs should be printed or suppressed. Under TSO, **execmsi** indicates whether PROFILE MSGID or PROFILE NOMSGID is in effect. Under CMS, **execmsi** tests whether the CP EMSG setting is EMSG ON, EMSG TEXT, EMSG CODE, or EMSG OFF.

Under OpenEdition, **execmsi** always returns **MSI_MSGON**, indicating that message IDs should be printed.

RETURN VALUE

The **execmsi** function returns an integer value indicating the user's message processing preference. Symbolic names for these return values are defined in the header file **<exec.h>**, as follows: MSI_MSGON specifies to print message and message id (TSO PROFILE MSGID or CMS SET EMSG ON or OpenEdition). MSI_MSGTEXT specifies to print message text only (TSO PROFILE NOMSGID or CMS SET EMSG TEXT). MSI_MSGCODE specifies to print message ID only (CMS SET EMSG OFF). MSI_MSGOFF specifies not to print messages (CMS SET EMSG OFF). MSI_NOINFO specifies that the information is unavailable (MVS batch).

EXAMPLE

This example formats a message to **stderr** based on the return code from **execmsi**:

```
#include <stdio.h>
#include <string.h>
#include <exec.h>

void msgfmt(int msgid, char *msgtext)
{
   int rc;
   rc = execmsi();
   switch (rc)
   {
      case MSI_MSGON:       /* id + text  */
      default:
         fprintf(stderr, "%d -- %s\n", msgid, msgtext);
         break;
      case MSI_MSGTEXT:     /* text only  */
         fprintf(stderr, "%s\n", msgtext);
         break;
```

execmsi Return System Message Preference
(continued)

```
            case MSI_MSGCODE:     /* id only    */
               fprintf(stderr, "%d\n", msgid);
               break;
            case MSI_MSGOFF;      /* no message */
               break;
         }
         return;
      }
```

execrc Set Return Code of Most Recent Subcommand

SYNOPSIS

```
#include <exec.h>

int execrc(int rc);
```

DESCRIPTION

The **execrc** function is used to set the return code of the most recently executed subcommand. The return code is passed back to any executing EXEC or CLIST and is accessible via the REXX RC variable, the EXEC2 &RC variable, or the CLIST &LASTCC variable. In all environments, a negative return code is treated as a more serious error than a positive code.

RETURN VALUE

The **execrc** function returns the maximum return code value that is specified so far. That is, the return value is the maximum of the current **execrc** argument and any previous argument to **execrc**. Under TSO and OpenEdition, the return code is replaced by its absolute value before the maximum is computed.

execrc returns a negative value if it is unable to successfully store the requested return code.

CAUTIONS

If **execrc** is called more than once for the same subcommand, the last value set is used, but the **execrc** return value may reflect the previous value.

If **execrc** is not called for each subcommand, the old value of **rc** is retained.

If **execrc** is not called at all for a subcommand, the return code for the previous subcommand is set, or **0** is used if there is no previous value.

IMPLEMENTATION

Under TSO, if **rc** is negative, the IKJSTCK service routine is called to flush the stack. Then, the absolute value of **rc** is stored in the ECTRCDF field, which the CLIST processor uses as the value of the &LASTCC variable.

However, a call to **execrc** with a negative argument does not flush a REXX EXEC because this functionality is not present in TSO REXX. In this case, the EXEC receives **rc** unchanged, not its absolute value, as for a CLIST. If **execrc** is called with a negative argument and one or more CLISTs are active, the CLISTs are still flushed until a REXX EXEC or the terminal is at the top of the stack, at which point flushing stops.

Under CMS, the **rc** value is the R15 value returned when control is returned to CMS, presumably in search of a new subcommand.

EXAMPLE

See the example for the **execget** function and the comprehensive SUBCOM example.

execshv Fetch, Set, or Drop REXX, EXEC2, or CLIST Variable Values

SYNOPSIS

```
#include <exec.h>

int execshv(int code, char *vn, int vnl, char *vb,
            int vbl, int *vl);
```

DESCRIPTION

Under TSO or CMS, the **execshv** function can be called in any situation where an EXEC or CLIST is active. Under OpenEdition, **execshv** can be used only when an EXEC invoked by **execcall** is active.

execshv performs the following tasks:

- copies the value of a REXX, EXEC2, or CLIST variable to an array of **char**
- assigns a string value to a new or existing REXX, EXEC2, or CLIST variable
- undefines (drops) a CMS or OpenEdition REXX variable or stem, or causes a CLIST or TSO REXX variable to be assigned a null (length 0) value.

The value of **code** determines what action **execshv** performs and how the remaining parameters are used. **<exec.h>** defines the following values that can be used as the value of **code**:

SHV_DROP
drops the named REXX or CLIST variable if it exists. The name is translated to uppercase before being used. For TSO CLIST or REXX variables, the variable name cannot truly be dropped; instead, it is set to a zero-length string.

SHV_FETCH
fetches the value of a REXX, EXEC2, or CLIST variable to a buffer, **vb**. The variable name is translated to uppercase before being used.

SHV_FIRST
initializes the **SHV_NEXT** fetch next sequence and retrieves the first in the list of all names and values of all REXX or CLIST variables as known to the REXX or CLIST interpreter. The particular variable fetched is unpredictable.

SHV_NEXT
fetches the next in the list of names and values of all REXX or CLIST variables as known to the REXX or CLIST interpreter. The order in which variable names and values are fetched is unpredictable. The fetch loop can be reset to start again by calling **execshv** with **SHV_FIRST** or any other value for **code**.

SHV_SET
sets a REXX, EXEC2, or CLIST variable to a value. The case of the variable name, **vn**, is ignored (no distinction is made between uppercase and lowercase characters).

The values of the remaining arguments **vn**, **vnl**, **vb**, **vbl**, and **vl** are controlled by **code.** The values of these arguments are summarized in **Table 23.1** .

execshv *Fetch, Set, or Drop REXX, EXEC2, or CLIST Variable Values (continued)*

Table 7.1 *execshv Argument Values*

code	vn	vnl	vb	vbl	vl
SHV_SET	addresses the name of the variable	length of the variable name. If 0, then the name is assumed to be null-terminated.	addresses a buffer containing the value to be assigned	length of the value. If 0, then the value is assumed to be null-terminated.	ignored
SHV_FETCH	addresses the name of the variable	length of the variable name. If 0, then the name is assumed to be null-terminated.	addresses a buffer to which the value of the variable is copied	length of the buffer	addresses an `int` where the actual length of the value will be stored. If null, then the value will be null-terminated.
SHV_FIRST SHV_NEXT	addresses a buffer where the name is copied. The name returned is null-terminated.	length of the variable name buffer	addresses a buffer to which the value of the variable is copied	length of the buffer	addresses an `int` where the actual length of the value will be stored. If null, then the value will be null-terminated
SHV_DROP	addresses the name of the variable	length of the name. If 0, the name is assumed to be null-terminated.	ignored	ignored	ignored

RETURN VALUE

The **execshv** function returns a negative value if the operation cannot be performed and a nonnegative value if it is performed. **<exec.h>** defines the negative values from **execshv** as follows:

SHV_NO_SUBCOM
specifies that under CMS, neither REXX, EXEC2, nor any subcommand environment is active. Under TSO, no CLIST or REXX EXEC is active. Under OpenEdition, no SUBCOM environment is active or no REXX exec is active.

SHV_NO_MEM
specifies that there is not enough memory available to complete the operation.

SHV_LIBERR
specifies an **execshv** internal library error.

execshv Fetch, Set, or Drop REXX, EXEC2, or CLIST Variable Values
(continued)

SHV_INVALID_VAR
specifies that the variable name or value is invalid. The name does not adhere to environmental restrictions (for example, the name contains invalid characters, the length is less than 0, the length is greater than 250 for REXX or 252 for TSO CLIST, and so on) or the value is invalid (for example, longer than 32K bytes in a TSO CLIST environment).

SHV_INVALID_FUNC
specifies that the function code is not one of **SHV_SET**, **SHV_FETCH**, **SHV_DROP**, **SHV_FIRST**, or **SHV_NEXT**.

SHV_NOT_SUPPORTED
specifies that the current environment does not support or have the TSO CLIST variable interface module IKJCT441. This module is supplied with TSO/E Version 1 Release 2.1 or higher systems.

SHV_SIGNAL
specifies that the REXX interface process was terminated by an OpenEdition signal.

Nonnegative return values from **execshv** are any one or more of the following. When one or more of these conditions are true, a logical OR is performed to indicate that the condition occurred.

SHV_SUCCESS
specifies that the operation completed successfully.

SHV_NOT_FOUND
specifies that the variable name was not found.

SHV_LAST_VAR
is returned after all variables from a **SHV_NEXT** loop have been fetched, as in an end-of-file return. The returned variable name and value are unpredictable.

SHV_TRUNC_VAL
specifies that for **SHV_FETCH**, **SHV_FIRST**, or **SHV_NEXT**, the buffer addressed by **vb** is too short to contain the entire value of the variable. If **vl** addresses an **int**, the actual length of the value is stored in ***vl**.

SHV_TRUNC_VAR
specifies that for **SHV_FIRST** or **SHV_NEXT**, this value may be returned if the buffer addressed by **vn** is too short to contain the entire variable name.

IMPLEMENTATION

Under CMS, **execshv** uses the direct interface to REXX and EXEC2 variables provided by EXECCOMM. Under TSO, **execshv** uses the IKJCT441 interface to CLIST and REXX variables. Under OpenEdition, **execshv** uses the IRXEXCOM service.

USAGE NOTES

The **<exec.h>** function also defines five macros for use with **execshv**: **shvset**, **shvfetch**, **shvdrop**, **shvfirst**, and **shvnext**. These macros SET, FETCH, DROP, FETCH FIRST in SEQUENCE, and FETCH NEXT in SEQUENCE, respectively, by expanding to the full form of the **execshv** function call to process REXX/EXEC2 or CLIST variables. Using these macros

execshv Fetch, Set, or Drop REXX, EXEC2, or CLIST Variable Values *(continued)*

can, in some situations, simplify the use of **execshv** considerably. The definition of each macro is shown here, followed by an example of its use:

```
   /* shvset macro */
#define shvset(vn, vb) execshv(SHV_SET, vn, 0, vb, 0, 0)

rc = shvset("XXXVAR","THIS_VALUE");

   /* shvfetch macro */
#define shvfetch(vn, vb, vbl)
    execshv(SHV_FETCH, vn, 0, vb, vbl, 0)

char valbuf[50];
rc=shvfetch("RVAR",valbuf,sizeof(valbuf));

   /* shvdrop macro */
#define shvdrop(vn) execshv(SHV_DROP, vn, 0, NULL, 0, 0)

rc = shvdrop("XXXVAR");

   /* shvfirst macro */
#define shvfirst(vn, vnl, vb, vbl)
    execshv(SHV_FIRST, vn, vnl, vb, vbl, 0)

   /*shvnest macro */
#define shvnext(vn, vnl, vb, vbl)
    execshv(SHV_NEXT, vn, vnl, vb, vbl, 0)

char vname[50], valbuf[255];
rc = shvfirst(vname,vnl,valbuf,sizeof(valbuf));
rc = shvnext (vname,vnl,valbuf,sizeof(valbuf));
```

EXAMPLE

The following program, named **shutstc.c**, demonstrates how to list all current REXX, EXEC2 or CLIST variables accessible to a program.

```
#include <exec.h>
#include <lcio.h>
#include <stdio.h>

main()
{
   int rc;
   int len;
   char namebuf[20];
   char valbuf[200];
```

execshv Fetch, Set, or Drop REXX, EXEC2, or CLIST Variable Values
(continued)

```
   rc = execshv(SHV_FIRST,namebuf,20,valbuf,200,&len);

   while (rc >= 0 && !(rc & SHV_LAST_VAR)) {
      if (rc & SHV_TRUNC_VAR) {
         puts("Variable name truncated.");
      }
      if (rc & SHV_TRUNC_VAL) {
         puts("Variable value truncated.");
         printf("Actual value length is %d\n",len);
      }
      printf("The variable name is: %s\n",namebuf);
      printf("The variable value is: %.*s\n",len,valbuf);
      rc = execshv(SHV_NEXT,namebuf,20,valbuf,200,&len);
   }
}
```

The following EXEC, which should be named **shutst.exec**, will declare several variables. The **shutstc.c** can be called to print these variables to the terminal.

```
Arnie = "cute"
Becca = "beautiful"
      .
      .
      .
'shutstc'
exit(0)
```

Examples of SUBCOM Processing

These examples demonstrate the SUBCOM interface to CMS and TSO. In the first example, three subcommands are accepted: ECHO repeats operands, SETRC sets a return code, and EXEC invokes a TSO CLIST or CMS EXEC. The program can be executed either interactively or noninteractively depending on the value of the **interact** option in **execinit**.

A copy of this example is provided with the compiler and library. See your SAS Software Representative for SAS/C compiler products for more information.

```
#include <exec.h>
#include <ctype.h>
#include <lcstring.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

main(int argc, char **argv)
{
      /* If 0, next input expected from EXEC.  If nonzero, next input */
      /* expected from terminal.                                      */
   int interact;

   char *input, *cmdname, *operands;
   int maxrc = 0;
   int thisrc;
   char msgbuff[120];
   int result;

   interact = (argc > 1 && tolower(*argv[1]) == 'i');

      /* If first argument starts with 'i', use interactive mode.     */
   result = execinit("EXAMPLE", interact);
   if (result  != 0) exit(EXIT_FAILURE);
      /* Check for failure of execinit */

   for(;;) {
      operands = input = execget();

         /* Obtain next input line. */
      if (!input)
         break;

      cmdname = execid(&operands);

         /* Obtain command name.  If error in execid, expect          */
         /* CLIST/EXEC input next.                                    */
      if (!cmdname) {
         interact = 1;
         continue;
      }

      strupr(cmdname);                  /* Upper case command name   */
```

```
if (!*cmdname)                    /* Check for null input.     */
   continue;

   /* If execid did an EXEC for us, note and continue.         */
else if (strcmp(cmdname, "*EXEC") == 0) {
   interact = 0;
   continue;
}

   /* If we just switched from EXEC to the terminal, note this. */
else if (strcmp(cmdname, "*ENDEXEC") == 0) {
   interact = 1;
   thisrc = atoi(operands);

      /* Extract EXEC return code. Remember it might be missing.*/
   if (operands)
      sprintf(msgbuff, "EXEC return code %d", thisrc);
   else
      strcpy(msgbuff, "EXEC return code unavailable");

      /* Inform user of return code. */
   result = execmsg("EXAM001I", msgbuff);
   if (result != 0 && errno == EINTR) break;
      /* Check for unexpected signal */
}

   /* Terminate if END received. */
else if (strcmp(cmdname, "END") == 0)
   break;

else if (strcmp(cmdname, "EXEC") == 0) {

      /* Call EXEC for EXEC subcommand and expect input from it.*/
   thisrc = execcall(input);
   interact = 0;
   if (thisrc != 0) {
      sprintf(msgbuff,"EXEC failed with return code %d",thisrc);
      result = execmsg("EXAM005E",msgbuff);
      if (result != 0 && errno == EINTR) break;
         /* just quit after OpenEdition signal */
   }
}

   /* If command is ECHO, repeat its operands.                 */
else if (strcmp(cmdname, "ECHO") == 0) {
   result = execmsg(0, operands);
   if (result != 0 && errno == EINTR) break;
   thisrc = 0;
}

   /* If command is SETRC, set return code as requested.     */
else if (strcmp(cmdname, "SETRC") == 0) {
   char *number_end;
   thisrc = strtol(operands, &number_end, 10);
```

```
         if (!number_end) {
            sprintf(msgbuff, "Invalid return code: %s",operands);
            result = execmsg("EXAM002E", msgbuff);
            if (result != 0 && errno == EINTR) break;
            thisrc = 12;
         }
         else {
            sprintf(msgbuff, "Return code set to %d", thisrc);
            execmsg("EXAM003I", msgbuff);
            if (result != 0 && errno == EINTR) break;
         }
      }

         /* If unknown command name, try to EXEC it.              */
      else {
         errno = 0;                  /* Make sure errno is clear   */
         thisrc = execcall(input);
         if (thisrc != 0 && errno == EINTR) break;
         interact = 0;
      }
      maxrc = execrc(thisrc);        /* Inform EXEC of return code.*/
      if (maxrc < 0 && errno == EINTR) break;
   }
   sprintf(msgbuff, "Maximum return code was %d", maxrc);
   execmsg("EXAM004I", msgbuff);     /* Announce max return code.  */
   execend();                        /* Cancel SUBCOM connection.  */
   return maxrc;                     /* Return max return code.    */
}
```

CLIST Example for SUBCOM Processing

The following CLIST illustrates the use of subcommand processing under TSO. The CLIST provides subcommands to the SAS/C SUBCMD sample.

```
CONTROL NOCAPS
WRITE SUBCOM EXAMPLE: starting.

   /* Issue the echo subcommand a few times. */
ECHO This is the first subcommand.
ECHO Your userid is &SYSUID..
SETRC 0     /* Try the SETRC subcommand. */
WRITE SUBCOM EXAMPLE: return code is now &LASTCC
SETRC 8     /* Set the return code to 8 */
WRITE SUBCOM EXAMPLE: return code is now &LASTCC
SETRC 0     /* and back to 0 */
WRITE

   /* Experiment with 'execmsg' by changing current       */
   /* message handling.  The PCF-II X facility is used to */
   /* direct PROFILE requests to TSO.                     */
X PROFILE MSGID    /* Request message ID display.         */
WRITE PROFILE MSGID (complete message)
SETRC 0
WRITE
X PROFILE NOMSGID  /* Request message ID suppression.     */
WRITE PROFILE NOMSGID (just the text)
```

```
SETRC 0
WRITE
CONTROL NOMSG
WRITE CONTROL NOMSG (no message)
SETRC 0
WRITE
CONTROL MSG
WRITE SET SYSOUTTRAP = 5 (trap messages within CLIST)
SET SYSOUTTRAP = 5
SETRC 0
SET I = 1
WRITE Output produced by SETRC subcommand:
DO WHILE &I <= &SYSOUTLINE    /* Write output of SETRC subcommand. */
   SET MSG = &STR(&&SYSOUTLINE&I)
   WRITE &MSG
   SET I = &I + 1
END
SET SYSOUTTRAP = 0
WRITE

   /* Tell the example program to invoke another CLIST,   */
   /* this one called SUBSUB.  When it finishes, display  */
   /* its return code.  SUBSUB returns the sum of its     */
   /* arguments as its return code.                       */
%SUBSUB 12 2
WRITE SUBCOM EXAMPLE: subsub returned &LASTCC.
END

   /* Invoked by SUBCOM CLIST */
PROC 2 VAL1 VAL2
CONTROL NOCAPS
WRITE SUBSUB EXAMPLE: Received arguments &VAL1 and &VAL2
EXIT CODE(&VAL1+&VAL2)
```

TSO REXX Example for SUBCOM Processing

The following REXX EXEC illustrates the use of subcommand processing with TSO REXX. The EXEC provides input to the SAS/C SUBCMD example.

```
address example               /* REXX SUBCOM example               */

   /* Issue the echo subcommand a few times.                        */
"echo This is the first subcommand."
"echo Your userid is " sysvar("SYSUID")
"setrc 0"                                /* try the SETRC subcommand */
say "SUBCOMR EXAMPLE: return code is now " RC
"setrc 8"                                /* Set the return code to 8 */
say "SUBCOMR EXAMPLE: return code is now " RC
"setrc 0"                                /* and back to 0            */
say " "
```

```
  /* Put an echo subcommand on the stack for execution after the EXEC */
  /* completes.                                                       */
queue 'echo This line is written after the EXEC is completed.'
  /* Experiment with "execmsg" by changing current message handling.  */
address tso "profile msgid"
say "PROFILE MSGID (complete message)"
"setrc 0"
say " "
address tso "profile nomsgid"
say "PROFILE NOMSGID (just the text)"
"setrc 0"
say " "
save = msg("OFF")
say "MSG(OFF) (no message)"
"setrc 0"
say " "
save = msg(save)

   /* Tell the example program to invoke a CLIST named SUBSUB. When   */
   /* it finishes, display its return code. SUBSUB returns the sum    */
   /* of its arguments as its return code.                            */
"%subsub 12 2"
say "SUBCOMR EXAMPLE: subsub returned " RC

   /* Now invoke a REXX EXEC named SUBSUBR, which performs the same   */
   /* function as the SUBSUB CLIST. Note that the negative return     */
   /* code produced by this call to subsubr will be flagged as an     */
   /* error by REXX.                                                  */
"%subsubr 5 -23"
say "SUBCOMR EXAMPLE: subsubr returned " RC
```

The following is the SUBSUBR EXEC invoked by the previous example.

```
   /* REXX EXEC invoked by SUBCOMR EXEC                               */
parse arg val1 val2 .
say "SUBSUBR EXAMPLE: Received arguments " VAL1 " and " VAL2
exit val1+val2
```

CMS EXEC Example for SUBCOM Processing

The following example illustrates the use of the SAS/C SUBCOM facility under CMS. Subcommands are provided to the SUBCOM SAS/C program. By default, the subcommands are sent to the environment with the same name as the macro filetype, in this case “EXAMPLE”.

```
   /*  REXX */
say 'SUBCOM EXAMPLE: starting.'
   /* Issue the echo subcommand a few times.                          */
'echo This is the first subcommand.'
'echo Your userid is' userid()'.'
'setrc 0'                          /* Try the setrc subcommand.     */
say 'SUBCOM EXAMPLE: return code is now' rc
                                   /* REXX variable 'rc'            */
'setrc 8'                          /* Set the return code to 8.     */
```

```
say 'SUBCOM EXAMPLE: return code is now' rc
                                    /* REXX variable 'rc'             */
'setrc 0'                           /* And back to 0.                 */
say ''

   /* Experiment with 'execmsg' by changing the current EMSG setting. */
   /* The subcommand 'setrc 0' will cause a message (or part of a     */
   /* message or nothing) to be sent. Note that the REXX 'address'    */
   /* statement is used to change the subcommand environment to CMS.  */
address  COMMAND 'CP SET EMSG ON'       /* Display both ID and text. */
say 'SET EMSG ON (complete message)'
'setrc 0'
say ''
address COMMAND 'CP SET EMSG CODE'      /* Display just the ID.      */
say 'SET EMSG CODE (just the id)'
'setrc 0'
say ''
address COMMAND 'CP SET EMSG TEXT'      /* Display just the text.    */
say 'SET EMSG TEXT (just the text)'
'setrc 0'
say ''
address COMMAND 'CP SET EMSG OFF'       /* Don't display anything.   */
say 'SET EMSG OFF (no message)'
'setrc 0'
say ''
address COMMAND 'CP SET EMSG ON'        /* Back to normal.           */

   /* Finally, tell SUBCOM C to invoke another macro, this one called */
   /* 'SUBSUB'. When it finishes, display the return code. SUBSUB     */
   /* returns the number of arguments it got as the return code.      */
'exec subsub arg1 arg2'
say 'SUBCOM EXAMPLE: subsub was passed' rc 'arguments.'
trace ?r
'end'
exit 0
   /* invoked by SUBCOM EXAMPLE                                       */
arg args
say 'SUBSUB EXAMPLE: Got' words(args) 'arguments.'
return words(args)
```

Guidelines for Subcommand Processing

Under CMS, programs receive an **``*ENDEXEC''** string indicating that an EXEC invoked externally has completed, and the program has been reentered during CMS end-of-command processing via the L$CEXEC nucleus extension (see the **execinit** description earlier in this chapter). Noninteractive execution of SUBCOM applications cannot be supported in bimodal CMS because, by the time the the library's end-of-command nucleus extension is entered, CMS has already purged program modules from memory.

Under the OpenEdition shell, the SAS/C REXX interface runs as a separate process from the calling program to prevent interference between the program and EXEC processing. Since the REXX interface is in a separate process, it is possible that the interface could be terminated by a signal at any time. When this occurs, on the next call to a SUBCOM function, **`errno`** is set to **`EINTR`**. Because the REXX interface has been terminated, the program cannot thereafter use any SUBCOM functions other than **`execend`**. Once **`execend`** has been called, it is possible to call **`execinit`** to create a new SUBCOM environment.

8 The CMS REXX SAS/C® Interface

8-1 Introduction
8-1 REXX Concepts and Background
8-1 Extending REXX with Function Packages
8-3 How the Library Interfaces with REXX
8-3 The SAS/C Library REXX Interface Functions
8-4 The SAS/C Library Function Package Support
8-4 C Function Packages and Nucleus Extensions
8-5 The SAS/C Library Interface and the REXX Extended Plist
8-5 When REXX Calls a C Function Package
8-6 Function Descriptions
8-17 An Example of a REXX Function Package
8-19 Additional Guidelines and Related Topics
8-19 Developing C Function Packages
8-19 Linking REXX Function Packages
8-20 Using the SAS/C Debugger and the IC Command

Introduction

This chapter covers the interface between the CMS REXX (Restructured Extended Executor) language and the SAS/C Library. The interface provided by the library enables you to extend REXX with function packages written in C language. Topics covered include an overview of REXX and function packages, a discussion of how the library interfaces with REXX, and a detailed discussion of the C functions involved. The chapter concludes with a comprehensive example and guidelines. This chapter is intended for applications and systems programmers who work in a CMS environment. A basic understanding of CMS and REXX is assumed.

REXX Concepts and Background

CMS provides an interpretive command and macro processor called the System Product Interpreter to interpret REXX programs. REXX is a high-level, general-purpose programming language. REXX is often used for writing EXECs (command procedures) or XEDIT macros. However, REXX is versatile enough for a wide range of additional programming applications.

As a programming language, REXX contains a large set of built-in functions. Using these functions makes it easier to write programs in REXX. However, in specialized situations, you may find that your program needs a function that REXX does not supply. For example, there is no built-in square root function. If you need a function that REXX does not provide, you can write specialized functions in other languages and group them together under a name recognized by REXX as referring to a *function package*. In this way you can extend the capabilities of your REXX program by calling these functions. Writing function packages in the C language for REXX programs becomes an efficient way to enhance REXX applications.

Extending REXX with Function Packages

Function packages add flexibility to the REXX language. For example, external functions can access REXX variables and return values to the REXX EXEC. In addition to extending the REXX language, function packages also can boost the performance of a REXX program. Because function packages are usually written in assembler or in a compiled language such as C, more complicated or arithmetically intensive functions can be executed in machine code rather than interpreted

word-by-word and line-by-line as for REXX. Also, function packages are loaded from disk only once, thereby avoiding the overhead of reloading each time a function is called.

Function packages are based on the programming concept of grouping sets of instructions (routines), designed to perform some specific task, outside of the mainline program. The REXX language allows calls to routines internal to the program and external to the program. Internal calls cause a branch to a routine identified by a statement label within the program. External calls are made to routines that reside in files outside both the user's program and the interpreter. Grouping similar external routines together into a function package is the focus of this chapter. For example, function packages typically are sets of related functions, such as **sin**, **cos**, and other trigonometric functions. Functions in a function package can share common code and data, or each function can be independent of the others.

The REXX interpreter recognizes three function package names. RXSYSFN is supplied with REXX and contains functions that interface with CP and CMS. The other two function packages are called RXUSERFN and RXLOCFN and can be written by any REXX user. Function packages written using the C language can use either of the names RXLOCFN or RXUSERFN.

To find and execute an external function, REXX goes through a specific search order. For example, if a REXX statement such as the following refers to a name that is not a label or the name of a built-in function, REXX searches for an external function with the name **csqrt**:

```
root = csqrt(100)
```

REXX searches for an external function by prefixing the function name with RX (RXCSQRT, in our example) and invoking it as a CMS command. If such a program is found, REXX invokes it with the argument list. However, if no such command can be found, REXX then searches for the function in either RXUSERFN or RXLOCFN.

Once called, a function in a function package can access its parameter list, get or set REXX variable values, and return a value to its caller just as any internal function can.

The next section covers the distinction between functions and subroutines, function packages as CMS nucleus extensions, and the parameter lists used between REXX and the function package routines.

Functions and subroutines

Within a REXX program, routines in a function package can be used either as functions or as subroutines. Because the C language does not have subroutines or procedures, this means partitioning functions into two types that REXX recognizes as functions or as subroutines. The distinction is that functions must return a result; subroutines need not return a result. A subroutine is called by the REXX CALL instruction. A function is called with a function call. For example, a function named **csqrt** would be called as **x = csqrt(4)**. To use **csqrt** as a subroutine, the call would be **call csqrt(4)**.

REXX function packages as nucleus extensions

A REXX function package is an efficient use of code because it is not subject to repetitive reloading. This is because it resides in storage as a nucleus extension. The first time REXX invokes the package, the package copies itself into storage outside of the CMS user program area and identifies itself as a nucleus extension. Once it has been loaded and identified, the function package remains loaded until LOGOFF or until CMS is re-IPLed. Nucleus extensions come before MODULE files in the command search order, so the function package MODULE file is not reloaded as long as the nucleus extension is active. The function package also identifies each separate function as a nucleus extension entry point, thereby enabling REXX to call the function directly (after the first call) without going through the function package search again.

Function parameter lists and variable values

A special parameter list called an *Extended Plist* is used by REXX for function and subroutine calls. All external routines are invoked using this six-word plist. Word 5 of this plist points to a list of arguments for the function being invoked. These arguments are in the form of address/length pairs known as *Adlen pairs.* Adlen pairs also are used in the C function that builds and uses the REXX Extended Plist.

The use of Adlen pairs is related to the way REXX handles variable values. REXX keeps all its variable values, even numeric values, as character strings. Each variable value is kept internally as a pointer to a character string coupled with an **`int`** containing the length of the string. For example, given the following REXX statement, **`x`** is a string of length 3 with the value 100:

```
x = 100
```

REXX external functions must accept their parameter lists in this format and return values in this format to REXX. This is why function parameter lists are passed to the function in the form of an array of such Adlen pairs.

For more information about REXX interfaces to external functions, refer to the appropriate IBM documentation for your VM system.

How the Library Interfaces with REXX

The library takes advantage of all aspects of the REXX function package interface. This section explains how the library provides the functions that enable you to create function packages in C.

The SAS/C Library REXX Interface Functions

The library provides a set of functions that help you create and use REXX function packages. These functions and their purposes are as follows: **`cmsrxfn`** creates a function package by defining a set of C functions that can be called directly from REXX. **`rxeval`** returns a function result to REXX. **`rxresult`** returns a function result to REXX. **`cmsshv`** provides a way of assigning values to new and existing variables shared with REXX or of dropping REXX variables or stems. (REXX stems allow collections of variables to be handled.) The following three macros are provided with this function:

`execset`
sets a REXX variable. **`execfetch`** fetches a REXX variable. **`execdrop`** drops a REXX variable.

cmsstack
inserts a string into the CMS program stack (the system-provided data queue that can be shared by REXX and your program). The following two macros provide stack access as well:

cmspush	stack (LIFO) access.
cmsqueue	queue (FIFO) access.

You use these functions in your program as follows:

- Code a **main** function that uses **cmsrxfn** to define all the functions in the function package. You do not need to have the **main** function return a value to CMS because **cmsrxfn** sets these return values.
- Code the individual functions for the package. If the function or subroutine returns a result, use the **rxeval** or **rxresult** function to set the value of the REXX RESULT variable. The following differences between functions and subroutines should be considered in your program:
  - If the REXX program can call the function as a true function, be sure the C function returns a value.
  - If the REXX program only calls the C function as a subroutine using a CALL statement, the C function does not need to return a value.
- Use the **cmsshv** and **cmsstack** functions and the **execset**, **execfetch**, **execdrop**, **cmspush**, and **cmsqueue** macros as needed in the function package.

The SAS/C Library Function Package Support

In addition to these functions, the library defines a special C program entry point, REXXMAIN. Together with **cmsrxfn**, REXXMAIN provides the interface that supports the use of C programs as REXX function packages. Together, REXXMAIN and **cmsrxfn** provide all of the support necessary to

- load the C program as a nucleus extension
- identify C functions as REXX external functions
- transfer control and parameter lists from REXX to C functions
- transfer control and result values from C functions back to REXX
- handle special situations under CMS, such as the NUCXDROP command and ABENDs.

When a C program is entered via REXXMAIN, the REXXMAIN code copies the program into storage and identifies it as a nucleus extension by calling the CMS command NUCXLOAD for itself. REXXMAIN then transfers control to the normal C initialization routine, passing a specially formatted parameter list. The C environment is created normally, and the parameter list created by REXXMAIN is passed to the **main** function as the argument array **argv**. (Note that REXX function packages are slightly unusual because they are always invoked by REXX instead of from the CMS command line.)

The C program then calls the **cmsrxfn** function, passing it the **argv** array and an array of pointers to C functions. The C functions pointed to by this array are made available to REXX as external functions. **cmsrxfn** thereafter handles the transfer of control and data between REXX and C. If a special situation occurs, such as an ABEND under CMS, **cmsrxfn** returns to its caller, which can then do such cleanup as required.

C Function Packages and Nucleus Extensions

During the execution of a REXX function package written using the library interface, several nucleus extension entry points are created. Two have the same name as the function package. The first of these nucleus extension entry points has the SYSTEM attribute and owns the storage occupied by the package code. The entry point identifies the location where the package should be entered if the C environment has

been destroyed by an ABEND and must be reinitialized. This support ensures that the C program remains in storage even if an ABEND occurs under CMS. Thus, the function package does not need to be reloaded from disk. The C environment, even though destroyed by the ABEND, is reinitialized automatically the next time the function package is called. The second nucleus extension entry point has the SERVICE attribute. Its entry point identifies the location within **cmsrxfn** that REXX enters on the second and subsequent calls to the function package.

A nucleus extension entry point is created for each C function called by REXX. The name is the name of the C function prefixed with **RX**. All of these entry points identify the same location within **cmsrxfn**. When REXX calls the function directly, this entry point is responsible for restoring the C environment and transferring control to the C function.

Between calls to the function package (after a function returns to REXX), the C environment is saved. Thus, open files remain open, memory stays allocated, and external variables retain their values.

For more information on the SYSTEM and SERVICE command attributes and nucleus extensions, refer to the CMS command and macro reference appropriate for your release.

The SAS/C Library Interface and the REXX Extended Plist

When a C function is called by REXX, the first parameter is always a pointer to an array of Adlen pairs. Each Adlen pair describes an argument to the function. The REXX_PLIST structure can reference members of this array. This structure, contained in the **<cmsexec.h>** header file, is defined as follows:

```
struct REXX_PLIST {
   char *ad;
   int len;
};
```

The variable **ad** points to an argument string, and **len** provides the length. Together these fields are used by REXX to map the elements in a REXX argument array (Adlen pairs). Omitted function arguments are denoted by a **NULL** value in the **ad** element. To see how these arguments are used with the REXX plist, refer to “cmsrxfn” on page 8-7.

When REXX Calls a C Function Package

When REXX calls one of the functions in a C function package, REXX creates the six-word plist and invokes the function as a command. Because the nucleus extension entry point indicates a location inside **cmsrxfn**, **cmsrxfn** is re-entered. **cmsrxfn** then

- re-establishes the C environment
- determines whether the C function was called as a subroutine or as a function
- creates the parameter list (**args** and **subflag**) where **args** is a pointer to an array of Adlen pairs, and **subflag** is an integer that is nonzero when the function is called as a subroutine
- proceeds to call the function via the function pointer in the **fncv** array.

When the function returns, **cmsrxfn** checks the return value. If the return value is **0**, it checks to see if **rxeval** or **rxresult** was called within the same module in which **cmsrxfn** was called. If either result function was called within the same module in which **cmsrxfn** was called, **cmsrxfn** puts the value passed to **rxeval** or **rxresult** (via a special control block called an EVALBLOK) in word 6 of the REXX plist and then returns to REXX. The current C environment is then saved for the next call. (Note that if both **rxresult** and **rxeval** are called, **cmsrxfn** uses the value from the last call made.)

▶ ***Caution 370 mode***
Do not issue CMS commands between **`main`** and the call to **`cmsrxfn`** in 370 mode. The parameter list that is created by REXXMAIN and passed to **`main`** as the argument array **`argv`** is destroyed before it can be passed to **`cmsrxfn`**. ▲

Function Descriptions

Descriptions of each REXX SAS/C interface function follow. Each description includes a synopsis, a description, discussions of return values and portability issues, and an example. Also, errors, cautions, diagnostics, implementation details, and usage notes are included where appropriate. An additional example and discussion of a function package written in C follows the detailed function descriptions. The function package documentation concludes with additional guidelines on using REXX with the library interface.

cmsrxfn Create a REXX Function Package

SYNOPSIS

```
#include <cmsexec.h>
int cmsrxfn(int argc, const char *argv[], int fncc,
            REXX_FNC fncv[]);
```

DESCRIPTION

cmsrxfn defines a set of C functions that can be called directly from the System Product Interpreter (REXX). **fncc** specifies the number of functions that can be called in this manner, and **fncv** is a pointer to an array of function pointers. Each function pointer in the array points to a function of type REXX_FNC that can be called by REXX. The **argc** and **argv** parameters are the command-line parameters passed to the **main** function in the C function package. These parameters should not be altered before passing them to **cmsrxfn**.

RETURN VALUE

The nature of **cmsrxfn** is such that it does not return under usual circumstances. Therefore, **cmsrxfn** does not return the normal successful return code of **0**. If **cmsrxfn** cannot allocate enough storage for control blocks or cannot install the module as a nucleus extension, it returns **-1**. If the module is terminated by an ABEND under CMS, **cmsrxfn** returns **1**. If the module is terminated by a NUCXDROP command, **cmsrxfn** returns **2**.

CAUTIONS

REXX typically calls a function package with a parameter list of the following form:

```
RXLOCFN LOAD FUNC1
```

This parameter list is passed to the **main** function via **argc** and **argv**. These parameters should be passed directly to **cmsrxfn** without modification.

REXX calls a function in a function package with a parameter list in the form of an array of pairs of character pointers and integers, each pair describing a parameter. (Refer to "The SAS/C Library Interface and the REXX Extended Plist" on page 8-5). All parameters, including numbers, are in character format. The array is terminated with a **char ***, **int** pair where the value of the **char *** is REXX_LAST_AD (defined in **<cmsexec.h>**), and the value of the **int** is REXX_LAST_LEN.

The result value, assigned to the REXX variable RESULT, also should be in character format.

If a C function called as a REXX function (identified as such in the array pointed to by **fncv**) returns a nonzero value via the RETURN statement, REXX ignores any RESULT value returned by the function. To set the REXX variable RC, use **cmsshv**. To assign a value to the REXX variable RESULT or to return a value from a C function called as a REXX function, use **rxresult** or **rxeval**.

cmsrxfn Create a REXX Function Package
(continued)

IMPLEMENTATION

cmsrxfn causes the program to be installed as a REXX function package.

EXAMPLE

```
#include <cmsexec.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

static int list();                  /* Declare REXX external function.*/
REXX_FNC rxfncs[] = {& list};

void main(int argc, char *argv[] )
{
   int rc;
      /* Call cmsrxfn, passing the argc, argv parameters, the number  */
      /* of functions to define (1), and an array of pointers to the  */
      /* function(s).                                                 */
   rc = cmsrxfn(argc, argv, 1, rxfncs);
   printf("cmsrxfn completed with return code %d\n",rc);
}

   /* This REXX function types a note indicating whether it was       */
   /* called as a function or as a subroutine (via a REXX call        */
   /* statement ). It then lists the arguments it was called with     */
   /* and sets the value of the REXX variable RESULT to the number    */
   /* of arguments.                                                   */
static list(args, subflag)
struct REXX_PLIST args[];
int subflag;
{
   int n;                           /* number of arguments passed     */
   char result_buffer[3];           /* buffer for REXX RESULT string  */

   if (subflag)      /* Were we called as a subroutine or a function? */
      puts("Called as a subroutine.");
   else
      puts("Called as a function.");
      /* Count arguments and print.  REXX will provide up to ten      */
      /* argument strings.                                            */
   for (n = 0; args[n].len != REXX_LAST_LEN; n++)
      if (args[n].ad != NULL)
         printf("Argument %d: \"%.*s\"\n",
                n, args[n].len, args[n].ad);
```

cmsrxfn Create a REXX Function Package
(continued)

```
      else
         printf("Argument %d: (Omitted)\n", n);
   if (n == 0) puts("No arguments passed.");
      /* Convert 'n' to string and set REXX RESULT variable.     */
   sprintf(result_buffer, "%d", n);
   rxresult(result_buffer);
   return 0;
}
```

cmsshv Fetch, Set, or Drop REXX or EXEC2 Variable Values

SYNOPSIS

```
#include <cmsexec.h>
int cmsshv(int code, char *vn, int vnl, char *vb,
           int vbl, int *vl);
```

DESCRIPTION

cmsshv can be called in any situation where REXX or EXEC2 is used, as well as in conjunction with function packages. **cmsshv** performs the following tasks:

- copies the value of a REXX or EXEC2 variable to an array of **char**
- assigns a string value to a new or existing REXX or EXEC2 variable
- undefines (drops) a REXX variable or stem.

The value of **code** determines what action is performed by **cmsshv** and how the remaining parameters are used. **<cmsexec.h>** defines the following values that may be used as the value of **code**:

SHV_SET_DIRECT
sets a REXX or EXEC2 variable name to a value. The variable name, **vn**, must be uppercase. If the name is a REXX stem, the characters following the period can be in mixed case.

SHV_SET_SYM
sets a REXX variable name to a value. The name can be in mixed case. EXEC2 does not support this value for **code**.

SHV_FETCH_DIRECT
fetches the value of a REXX or EXEC2 variable to a buffer, **vb**. The variable name must be uppercase. If the name is a REXX stem, the characters following the period can be in mixed case.

SHV_FETCH_SYM
fetches the value of a REXX variable to a buffer. The name can be in mixed case. EXEC2 does not support this value for **code**.

SHV_FETCH_PRIV
fetches REXX private information to a buffer. The variables recognized are

- **ARG**, which fetches the argument string that is parsed by the REXX statement PARSE ARG
- **SOURCE**, which fetches the source string that is parsed by the REXX statement PARSE SOURCE
- **VERSION**, which fetches the version string that is parsed by the REXX statement PARSE VERSION.

SHV_FETCH_NEXT
fetches the names and values of all REXX variables as known to the REXX interpreter. The order in which the variable names and values are fetched is unpredictable. The fetch loop can be reset to start again by calling **cmsshv** with any other value for **code**.

SHV_DROP_DIRECT
drops the named REXX variable, if it exists. The name must be in uppercase or, if it is a stem, all characters preceding the period must be in uppercase. If the name is a stem, then all variables beginning with that stem are dropped.

cmsshv Fetch, Set, or Drop REXX or EXEC2 Variable Values
(continued)

SHV_DROP_SYM
behaves the same as **SHV_DROP_DIRECT** except that the name can be in mixed case.

The values of the remaining arguments **vn**, **vnl**, **vb**, **vbl**, and **vl** are controlled by **code**. The values of these arguments are summarized in the following table.

Table 8.1 *cmsshv Argument Values*

code	vn	vnl	vb	vbl	vl
SHV_SET_DIRECT SHV_SET_SYM	addresses the name of the variable	length of the variable name; if 0, then the name is assumed to be null-terminated	addresses a buffer containing the value to be assigned	length of the value; if 0, then the value is assumed to be null-terminated	ignored
SHV_FETCH_DIRECT SHV_FETCH_SYM SHV_FETCH_PRIV	addresses the name of the variable	length of the variable name; if 0, then the name is assumed to be null-terminated	addresses a buffer to which the value of the variable is copied	length of the buffer	addresses an **int** where the actual length of the value will be stored. If **NULL**, then the value will be null-terminated
SHV_FETCH_NEXT	addresses a buffer where the name is copied. The name returned is null-terminated.	length of the variable name buffer	addresses a buffer to which the value of the variable is copied	length of the buffer	addresses an **int** where the actual length of the value will be stored. If **NULL**, then the value will be null-terminated
SHV_DROP_DIRECT SHV_DROP_SYM	addresses the name of the variable	length of the variable name; if 0, then the name is assumed to be null-terminated	ignored	ignored	ignored

cmsshv Fetch, Set, or Drop REXX or EXEC2 Variable Values
(continued)

RETURN VALUE

cmsshv returns a negative value if the operation could not be performed and a nonnegative value if it was performed. **<cmsexec.h>** defines the negative return values from **cmsshv** as follows:

SHVNOEXECCOMM
neither EXEC2 nor REXX is active.

SHVNOMEM
there is not enough memory available to complete the operation.

SHVLIBERR
cmsshv failed to create a correct EXECCOMM parameter list.

Nonnegative return values from **cmsshv** are any one or more of the following. When one or more of these conditions are true, they are logically OR'd to indicate that the condition occurred.

SHVSUCCESS
the operation completed successfully.

SHVNEWV
the variable name did not exist.

SHVLVAR
for SHV_FETCH_NEXT only, this is the last variable to be transferred.

SHVTRUNC
for SHV_FETCH_DIRECT and SHV_FETCH_SYM, the buffer addressed by **vb** was too short to contain the entire value of the variable. If **vl** addresses an **int**, the actual length of the value is stored in ***vl**. For SHV_FETCH_NEXT, this value can be returned if the buffer addressed by **vn** is too short to contain the entire name.

SHVBADN
the name of the variable is invalid.

SHVBADV
the value of the variable is too long. This value can be returned only when the value of **code** is SHV_FETCH_DIRECT and EXEC2 is active.

SHVBADF
the value of **code** is not one of the values defined in **<cmsexec.h>**.

IMPLEMENTATION

cmsshv uses the direct interface to REXX and EXEC2 variables (known as EXECCOMM) as documented in *VM/SP System Product Interpreter Reference*, IBM publication No. SC24-5239. Refer to this publication for a detailed explanation about this interface.

USAGE NOTES

<cmsexec.h> also defines three macros for use with **cmsshv**. The macros **execset**, **execfetch**, and **execdrop** assign, retrieve, and drop REXX variables, respectively. Using these macros can, in some situations, simplify the

cmsshv Fetch, Set, or Drop REXX or EXEC2 Variable Values *(continued)*

use of **cmsshv** considerably. Each macro is shown here, followed by an example.

```
   /* macro */
execset(char *vn, char *vb);
rc = execset("REXXVAR","THIS_VALUE");
   /* macro */
execfetch(char *vn, char *vb, int vbl);
char valbuf[50];
rc=execfetch("RVAR",valbuf,sizeof(valbuf));
   /* macro */
execdrop(char *vn);
rc = execdrop("REXXVAR");
```

EXAMPLE

```
#include <cmsexec.h>
#include <lcio.h>
#include <stdio.h>

main()
{
   int rc;
   int len;
   char namebuf[20];
   char valbuf[200];
   rc = cmsshv(SHV_FETCH_NEXT,namebuf,20,valbuf,200,& len);
   while (rc >= &&  !(rc & SHVLVAR)) {
      if (rc SHVTRUNC) {
         puts("Either name or value truncated.");
         printf("Actual value length is %d\n",len);
      }
      printf("The variable name is: %s\n",namebuf);
      printf("The variable value is: %.*s\n",len,valbuf);
      rc=cmsshv(SHV_FETCH_NEXT,namebuf,20,valbuf,200,& len);
   }
}
```

cmsstack Insert a String into the CMS Program Stack

SYNOPSIS

```
#include <cmsexec.h>

int cmsstack(int order, const char *str, int len);
```

DESCRIPTION

cmsstack inserts the character array addressed by **str** of length **len** onto the CMS program stack in either last-in-first-out (LIFO) or first-in-first-out (FIFO) order depending on the value of the **order** argument. (**cmsstack** can be used in any CMS application, not just with function packages.)

<cmsexec.h> defines two values for **order**: STK_LIFO and STK_FIFO. If **len** is **0**, then the character array addressed by **str** is assumed to be null-terminated.

RETURN VALUE

cmsstack returns **0** if the string was inserted or a nonzero value if the string was not inserted.

CAUTION

The maximum value of **len** (or if **len** is **0** the maximum length of the string addressed by **str**) is 255.

USAGE NOTES

<cmsexec.h> also defines two macros based on **cmsstack**. The definitions are shown here, followed by an example:

```
   /* cmspush */
#define cmspush(s) cmsstack(STK_LIFO, s, 0)

rc = cmspush("This string is stacked LIFO");

   /* cmsqueue */
#define cmsqueue(s) cmsstack(STK_FIFO, s, 0)

rc = cmsqueue("This string is stacked FIFO");
```

EXAMPLE

```
#include <cmsexec.h>

   /* Stack the parameter on the program stack in FIFO order.        */
int main(int argc, char *argv[])
{
   int rc;
   if (argc != 2)
      exit(8);
   rc = cmsstack(STK_FIFO,argv[1],0);
   exit(rc == 0 ? 0 : 8);
}
```

rxeval Return a Result Value to REXX

SYNOPSIS

```
#include <cmsexec.h>

int rxeval(const char *ptr, unsigned int len);
```

DESCRIPTION

rxeval assigns **len** bytes, starting at the location addressed by **ptr**, to the REXX variable RESULT.

rxeval is similar to the **rxresult** function except that the value assigned to the REXX variable RESULT can contain embedded NULL characters.

RETURN VALUE

rxeval returns **0** if the value was properly assigned or some nonzero value if the assignment fails.

CAUTIONS

rxeval can be used only in conjunction with the **cmsrxfn** function. If the return value from a function called from REXX is not **0**, then the value assigned by **rxeval** is ignored.

If both **rxresult** and **rxeval** are used in the same function, the last value assigned by either function is the value assigned to RESULT.

EXAMPLE

```
#include <cmsexec.h>
char hexdata[4];
int rc;

   /* Note that hexdata can contain any value */
rc = rxeval(hexdata,sizeof(hexdata));
```

rxresult Return a Result Value to REXX

SYNOPSIS

```
#include <cmsexec.h>

int rxresult(const char *str);
```

DESCRIPTION

rxresult assigns the string addressed by **str** to the REXX variable RESULT. (To set the REXX variable RC, use the **cmsshv** function.)

RETURN VALUE

rxresult returns 0 if the value was properly assigned or some nonzero value if the assignment fails.

CAUTIONS

rxresult can be used only in conjunction with the **cmsrxfn** function. If the return value from a function called from REXX is not 0, then the value assigned by **rxresult** is ignored.

If both **rxresult** and **rxeval** are used in the same function, the last value assigned by either function is the value assigned to RESULT.

EXAMPLE

An example of the use of **rxresult** is included in the **result** function of the sample program that follows these function descriptions.

An Example of a REXX Function Package

This example shows a REXX function package containing three trigonometric functions: **csqrt**, **csin**, and **ccos**. The routines in the package can be called either as functions or as subroutines from REXX.

A copy of this example program is provided with the compiler and library. See your SAS Software Representative for C compiler products for more information.

```
#include <cmsexec.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <options.h>

   /* Because this program cannot be invoked directly from the        */
   /* command line, run-time options should be specified via the      */
   /* '_options' variable. For example, int _options = _DEBUG;        */
static int csin(), ccos(), csqrt();
static double todouble();
static void result();

   /* Define the values of 'fncv' and 'fncc'.                         */
REXX_FNC funlist[] = {csin,ccos,csqrt};
#define NFUNCS sizeof(funlist)/sizeof(REXX_FNC)

void main(int argc, char *argv[]);
{
   int rc;
   rc = cmsrxfn(argc,argv,NFUNCS,funlist);
      /* A positive return code from cmsrxfn() indicates that either  */
      /* a NUCXDROP RXLOCFN was entered or an ABEND occurred under    */
      /* CMS.  A negative return code indicates that initialization   */
      /* did not complete.                                            */
   if (rc < 0)
      puts("RXLOCFN did not initialize.");
}

   /* Compute trigonometric sine.  Example:  x = csin(y)              */
static csin(struct REXX_PLIST args[]);
{
   register double r;
      /* Ensure that there is exactly one argument and that it is     */
      /* 15 or fewer characters long.  (Other validation is probably  */
      /* useful, but it has been omitted here.)                       */
   if (args->ad == REXX_LAST_AD || args->len > 15 ||
       args[1].ad != REXX_LAST_AD)
      return 1;
      /* Perform other parameter validation as necessary.             */
   r = todouble(args->ad,args->len);   /* Convert to double.          */
   r = sin(r);                         /* Get the sine.               */
   result(r);                          /* Set REXX 'result' variable. */
   return 0;                           /* Tell REXX it worked.        */
}
```

```
   /* Compute trigonometric cosine.  Example:  x= ccos(y)            */
static ccos(struct REXX_PLIST args[]);
{
   register double r;
   if (args->ad == REXX_LAST_AD || args->len > 15 ||
       args[1].ad != REXX_LAST_AD)
      return 1;
   r = todouble(args->ad,args->len);
   r = cos(r);
   result(r);
   return 0;
}

   /* Compute square root.  Example:  x = csqrt(y)                   */
static csqrt(struct REXX_PLIST args[]);
{
   register double r;
   if (args->ad == REXX_LAST_AD || args->len > 15 ||
       args[1].ad != REXX_LAST_AD)
      return 1;
   r = todouble(args->ad,args->len);
   if (r < 0.0)
      return 1;
   r = sqrt(r);
   result(r);
   return 0;
}

   /* Convert REXX parameter from Adlen to double.                   */
static double todouble(char *str, int len);
{
   char buff[16];                       /* Convert string to double. */
   double d;

      /* Copy to a temporary buffer and add a null terminator.       */
   memcpy(buff,str,len);
   buff[len] = '\0';
   d = strtod(buff,0);
   return d;             /* Return converted argument.              */
}

   /* Convert function result to char and set REXX result variable.  */
static void result(double r);
{
      /* Need enough room to handle leading 0, sign, decimal point,  */
      /* and exponent.                                               */
   char buff[15];
      /* This is similar to REXX's NUMERIC DIGITS 9 format.          */
   sprintf(buff,"%.9G",r);
   rxresult(buff);
}
```

Additional Guidelines and Related Topics

This section covers topics related to the REXX interface of the SAS/C Library. Included are additional guidelines for developing and using C function packages, linking function packages, and using the SAS/C Source Level Debugger with function packages.

Developing C Function Packages

The following notes should be considered when developing function packages:

- Function names are truncated to six characters, and lowercase letters are converted to uppercase. REXX prefixes the name with RX to form the eight-character command name.
- The **argv** array used by **cmsrxfn** is not a normal C **argv** array. It can be inspected but not modified.
- **cmsrxfn** expects functions to return normally via a **RETURN** statement. Therefore, **exit** and **longjmp** should not be used. Calls to these functions are trapped and cause an ABEND. Similarly, the REXXMAIN entry point also expects the main function to return by a **RETURN** statement. To comply with this condition, do not use **exit** to terminate a REXX function package.
- The function package is loaded in protected storage. Therefore, all REXX function packages must be compiled with the **RENT** compiler option. If the function package is not compiled with **RENT**, a diagnostic message is issued when the package is called, and it is not loaded.
- Very large function packages can be split into several dynamically loaded modules (see "loadm" on page 1-10). If you use dynamically loaded modules, all of the REXX function package functions (**cmsrxfn**, **rxresult**, **rxeval**) should be used only in the main load module. Using one of these functions in a dynamically loaded module will result in problems when the module is linked.
- REXX function packages written in SAS/C cannot be invoked recursively from REXX.

Linking REXX Function Packages

Refer to Chapter 6, "Compiling, Linking, and Executing Programs under CMS," in *SAS/C Compiler and Library User's Guide* for general information about how to link C programs. If you do not need to use COOL to preprocess the function package TEXT files, use the following CMS commands to create the function package MODULE. Assume that the function package is named RXLOCFN:

```
LOAD RXLOCFN (RLDSAVE RESET REXXMAIN
GENMOD RXLOCFN (FROM first_CSECT
```

Inspect the LOAD map produced by the LOAD command to determine the name of the first CSECT. The RLDSAVE option in the LOAD command causes the resulting MODULE to be relocatable. The RESET REXXMAIN option causes REXXMAIN to be the entry point for the MODULE. The FROM option forces the GENMOD command to save the MODULE, beginning with the first CSECT. Always use the name of the first CSECT in the object code as the FROM parameter. (The LOAD MAP file lists the names of the CSECTs in your program.)

If you have more than one compilation in your function package that initializes external variables, you need to use COOL to preprocess the TEXT files. You may also need to use COOL to remove pseudoregisters from the TEXT files. In this case, the following commands can be used to create the function package MODULE. (This assumes that the function package is created from several compilations named RXLOCP1, RXLOCP2, and so on.)

```
COOL RXLOCP1 RXLOCP2 ...
LOAD COOL (RLDSAVE RESET REXXMAIN
GENMOD RXLOCFN (FROM @EXTERN#
```

Do not use the GENMOD option of the COOL EXEC to produce the MODULE file. The COOL EXEC will not produce a relocatable MODULE. Instead, use a separate LOAD command to load the COOL370 TEXT file with the correct options. Note that, in this case, the first CSECT in the COOL370 TEXT file is @EXTERN#, which is used as the parameter to the FROM option in the GENMOD command.

Using the SAS/C Debugger and the IC Command

Function packages written in the C language can be debugged using the debugger just like any other C program.

Because you cannot specify the **=debug** run-time option via the command line, set the **_DEBUG** flag in the **_options** variable to start the debugger (see the previous example). When the function package is called the first time, the debugger initializes normally and prompts for a command. Use the debugger as you would in any other C program. When you finish debugging, recompile the function package without the **_DEBUG** flag set. (For more information about the **_options** variable, refer to the "Library Options" section of Chapter 9, "Run-Time Argument Processing," in *SAS/C Compiler and Library User's Guide*).

If the function package is active but no debugger commands are in effect or no debugger breakpoints are set, you can force the debugger to issue a prompt by entering IC on the CMS command line. The next time an EXEC calls a C function in the package, the debugger will issue a prompt.

9 Coprocessing Functions

9-1 Introduction
9-2 cocall and coreturn
9-3 Coprocess States
9-4 Passing Data between Coprocesses
9-5 Special Cases
9-5 Coprocess Data Types and Constants
9-6 Coprocess Identifiers
9-6 Restrictions
9-6 A Coprocessing Example
9-9 Advanced Topics: Program Termination
9-9 Exit Processing for Secondary Coprocesses
9-10 Exit Processing for the Main Coprocess
9-10 Advanced Topics: Signal Handling
9-11 Function Descriptions

Introduction

Some large applications are most conveniently implemented as a set of communicating processes that run independently of each other except for occasional exchanges of messages or other information. The SAS/C coprocessing feature supports this sort of application in a natural, operating-system-independent way.

This feature enables you to implement a SAS/C program as several cooperative processes, or *coprocesses.* Each coprocess represents a single thread of sequential execution. Only one thread is allowed to execute at any particular time. Control is transferred from one coprocess to another using the **`cocall`** and **`coreturn`** functions; these functions also provide for the transfer of data between coprocesses. At the start of execution, a main coprocess is created automatically by the library. Additional coprocesses are created during execution using the **`costart`** function and are terminated via the **`coexit`** function so that the coprocess structure of a program is completely dynamic. Most of the data structures of a SAS/C program are shared among all coprocesses, including **`extern`** variables, storage allocated by **`malloc`**, and open files. (The structures that are not shared are listed later in this section.)

Note that coprocessing does not implement multitasking. That is, only a single coprocess can execute at a time. Furthermore, transfer of control is completely under program control. If the executing coprocess is suspended by the operating system to wait for an event, such as a signal or I/O completion, no other coprocess is allowed to execute. Finally, note that the implementation of coprocessing does not use any multitasking features of the host operating system.

Coprocesses are very similar to coroutines, as included in some other languages. The name coprocess was used rather than coroutine because many SAS/C functions can be called during the execution of a single coprocess.

cocall and coreturn

The **cocall** and **coreturn** functions are used to transfer control and information from one coprocess to another. Normally, the **cocall** function is used to request a service of a coprocess, and the **coreturn** function is used to return control (and information about the requested service) to a requestor. The transfer of control from one coprocess to another as the result of a call to **cocall** or **coreturn** is called a *coprocess switch.*

Use of **cocall** is very similar to the use of a normal SAS/C function call. They both pass control to another body of code, they both allow data to be passed to the called code, and they both allow the called code to return information. Whether you are using a function call or a **cocall**, after completion of the call the next SAS/C statement is executed.

In contrast, the SAS/C **return** statement and the **coreturn** function behave differently, even though both allow information to be returned to another body of code. Statements following a **return** statement are not executed because execution of the returning function is terminated. However, when **coreturn** is called, execution of the current coprocess is suspended rather than terminated. When another coprocess issues a **cocall** to the suspended coprocess, the suspended coprocess is resumed, and the statement following the call to **coreturn** begins execution.

The following example illustrates the way in which **cocall** and **coreturn** work. The two columns in the example show the statements to be executed by two coprocesses, labeled A and B. For simplicity, in this example no data are transferred between the two coprocesses.

Coprocess A

```
A_func()
{
   .
   . /* other statements */
   .
begin:
   puts("A1");
   cocall(B, NULL);
   puts("A2");
   cocall(B, NULL);
   puts("A3");
   .
   . /* other statements */
   .
}
```

Coprocess B

```
B_func()
{
   .
   . /* other statements */
   .
   coreturn(NULL);
   puts("B1");
   bsub();
   puts("B4");
   coreturn(NULL);
   .
   . /* other statements */
   .
void bsub()
{
      puts("B2");
      coreturn(NULL);
      puts("B3");
      return;
}
```

If coprocess A's execution has reached the label **begin**, and coprocess B is suspended

at the first **coreturn** call, then the following sequence of lines is written to **stdout**:

```
A1
B1
B2
A2
B3
B4
A3
```

Coprocess States

During its execution, a coprocess may pass through up to five states:

- starting
- active
- busy
- idle (or suspended)
- ended.

The effect of cocalling a coprocess depends on its state at the time of the call. (The function **costat** enables you to determine the state of a coprocess.)

You may find it helpful to trace through one of the examples in this section for concrete illustration of the state transitions described here.

A coprocess is created by a call to the **costart** function. After its creation by **costart**, a coprocess is in the *starting* state until its execution begins as the result of a **cocall**. (Because the main coprocess is active when program execution begins, it is never in the starting state.) Each coprocess has an *initial* function, which is specified as an argument to **costart** when the coprocess is created. This is the function that is given control when the coprocess begins execution.

A coprocess is *active* when its statements are being executed. When a coprocess is cocalled, it becomes active if the cocall was legal. An active process cannot be cocalled; an attempt by a coprocess to cocall itself returns an error indication. At the start of execution, the main coprocess is in the active state.

A coprocess is *busy* when it has issued a **cocall** to some other coprocess and that coprocess has not yet performed a **coreturn** or terminated. A busy coprocess cannot be cocalled, and an attempt to do so returns an error indication. This means that a coprocess cannot cocall itself, directly or indirectly. One implication of this rule is that it is not possible to cocall the main coprocess.

A coprocess becomes *idle* (or *suspended*) when it has called **coreturn**, if the call was legal. An idle coprocess remains idle until another coprocess cocalls it, at which point it becomes active. The main coprocess is not permitted to call **coreturn** and, therefore, cannot become idle.

A coprocess is *ended* after its execution has terminated. Execution of a coprocess is terminated if it calls the **coexit** function, if its initial function executes a **return** statement, or if any coprocess calls the **exit** function. In the last case, all coprocesses are ended. In the other two cases, the coprocess that cocalled the terminating coprocess is resumed, as if the terminated coprocess had issued a **coreturn**. You cannot cocall a coprocess after it has ended.

The effect of the various routines that cause coprocess switching can be summarized as follows:

cocall can be used to resume a starting or idle coprocess. The active coprocess becomes busy and the cocalled coprocess becomes active.

coreturn can be used to suspend the active coprocess and resume the one that cocalled it. The active coprocess becomes idle, and the one that cocalled it changes from busy to active.

coexit can be used to terminate the active coprocess and resume the one that cocalled it. The active coprocess becomes ended, and the one that cocalled it changes from busy to active.

Passing Data between Coprocesses

The **cocall**, **coreturn**, and **coexit** functions all provide an argument that can be used to pass data from one coprocess to another. Because the type of data that coprocesses may want to exchange cannot be predicted in advance, values are communicated by address. The arguments and returned values from these functions are all defined as the generic pointer type **void ***.

Except when a coprocess is beginning or ending execution, a call to either **cocall** or **coreturn** by one coprocess causes the calling coprocess to be suspended and a call to the other function in another coprocess to be resumed. In either case, the argument to the first function becomes the return value from the function that is resumed.

To illustrate, here is a version of the previous example that demonstrates how data can be passed from coprocess to coprocess:

Coprocess A

```
A_init()
{
   char *ret;
   B = costart(&B_init,NULL);
   ret = cocall(B, "B0");
   puts(ret);
   ret = cocall(B, "B1");
   puts(ret);
   ret = cocall(B, "B2");
   puts(ret);
}
```

Coprocess B{mono

```
char * B_init(arg)
char *arg;
{
   puts(arg);
   arg = coreturn("A1");
   puts(arg);
   puts(bsub());
   return "A3"; /* or coexit("A3"); */
}

char *bsub()
{
   char *arg;
   arg = coreturn("A2");
   return arg;
}
```

When the function **A_init** is called, the following sequence of lines is written to **stdout** :

```
B0
A1
B1
A2
B2
A3
```

Each line is written by the coprocess indicated by its first letter.

Special Cases

In the case where a **cocall** causes execution of a coprocess to begin or where a return from the initial function of a coprocess causes it to terminate, the simple rule that the argument to one function becomes the return value from another does not apply. In the first case, the argument to **cocall** becomes the argument to the initial function of the newly started coprocess. In the second case, the return value from the initial function becomes the value returned by the **cocall** in the resumed coprocess. Both of these situations are illustrated in the preceding example.

Coprocess Data Types and Constants

The header file **<coproc.h>** must be included (via a **#include** statement) in any compilation that uses the coprocessing feature. In addition to declaring the coprocessing functions, the header file also defines certain data types and constants that are needed when coprocesses are used. These definitions from **<coproc.h>** are as follows:

```
typedef unsigned coproc_t;

   /* cocall/coreturn error code */
#define CO_ERR (char *) -2

   /* coproc argument values */
#define MAIN 0
#define SELF 1
#define CALLER 2

   /* costat return values -- coprocess states */
#define ENDED 0
#define ACTIVE 1
#define BUSY 2
#define IDLE 4
#define STARTING 8
```

The data type **coproc_t** defines the type of a coprocess identifier. Coprocess IDs are described in more detail in “Coprocess Identifiers” on page 9-6 .

The value **CO_ERR** is returned by **cocall** and **coreturn** when the requested function cannot be performed. You should avoid passing this value as an argument to **cocall** or **coreturn**, because it can be misinterpreted by the paired function as indicating a program error rather than a correct return value. Note that **NULL** can be used as a **cocall** or **coreturn** argument. This is the recommended way to signify no information.

<coproc.h> also defines the type **struct costart_parms**, which is a structure containing start-up information for a coprocess, such as an estimate of the required stack size. The address of a structure of this type is passed to **costart** when a new coprocess is created. See “costart” on page 9-23 for more information on the uses for this structure.

Coprocess Identifiers

Coprocess identifiers are values of type **coproc_t**. They are returned by the **costart** function when a coprocess is created and passed to **cocall** to indicate the coprocess to be resumed. **0** is not a valid coprocess ID; this value is returned by **costart** when a new coprocess cannot be created.

Each coprocess has a unique ID value; the IDs of ended coprocesses are not reused.

The IDs of specific coprocesses can be found by calling the **coproc** function. See "coproc" on page 9-17 for details.

Restrictions

You should be aware of the following restrictions and special considerations when coprocesses are used. In addition, certain advanced topics that are important to a few complex applications are described here, after the example. You may want to skip these sections if they are not relevant to your application.

- At most, 65,536 coprocesses can be defined at one time, including the main coprocess. Coprocesses that have ended are not included in this count.
- Each coprocess has its own stack and automatic storage. The initial stack allocation run-time option applies only to the main coprocess. All other coprocesses are given an initial stack allocation of 4096 bytes unless some other size is requested by the call to **costart**. However, each coprocess' stack space is expanded dynamically as necessary.
- You may not use the minimal form of program linkage (the **=minimal** run-time option or its compile-time equivalent) in an application that uses coprocesses.
- Each coprocess has its own quiet setting. A call to the **quiet** function in one coprocess has no effect on diagnostic messages generated by another.
- Signal handling in a coprocessing environment requires special care. See "Advanced Topics: Signal Handling" on page 9-10 for more information.
- If any coprocess calls the **exit** function, the entire program is terminated. The details of program termination are described in "Advanced Topics: Program Termination" on page 9-9 . A call to **coexit** in the main coprocess is treated as a call to **exit**.
- **longjmp** cannot be used to transfer control between coprocesses. In particular, you cannot call **setjmp** in one coprocess and call **longjmp** using the same **jmp_buf** in another. Any attempt to do this causes abnormal program termination.
- Memory allocated via **malloc** is shared by all coprocesses. Memory allocated by one coprocess can be freed by another. Similarly, load modules loaded by one process can be used or unloaded by another, and files opened by one coprocess can be used or closed by another.
- There is no restriction against using the same function as the initial function of several coprocesses. In fact, as the following example shows, this can be a useful technique.

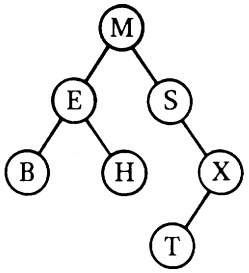
A Coprocessing Example

The following example shows a programming problem that is difficult to solve without coprocesses but is easy when coprocesses are used.

A convenient data structure for many applications is the sorted binary tree, which is a set of items connected by pointers so that it is easy to walk the tree and return the items in sorted order. A node of such a tree can be declared as **`struct node`**, defined by the following:

```
struct node {
   struct node *before;
   struct node *after;
   char *value;
};
```

The following is an example of a sorted binary tree, with each node represented as a point and the before and after pointers represented as downward lines (omitting **`NULL`** pointers):



It is easy to write a C function to print the nodes of such a tree in sorted order using recursion, as follows:

```
prttree(struct node *head)
{
      /* Print left subtree.  */
   if (head->before) prttree(head->before);

      /* Print node value.    */
   puts(head->value);

      /* Print right subtree. */
   if (head->after) prttree(head->after);
}
```

Now, suppose it is necessary to compare two sorted binary trees to see if they contain the same values in the same order (but not necessarily the exact same arrangement of branches). Without coprocesses, this cannot easily be solved recursively because the recursion necessary to walk one tree interferes with walking the other tree. However, use of coprocesses allows an elegant solution, which starts a coprocess to walk each tree and then compares the results the coprocesses return. (Some error checking is

omitted from the example for simplicity.) The example* assumes that all values in the compared trees are strings and that **NULL** does not appear as a value.

```
#include <stddef.h>
#include <coproc.h>
#include <string.h>

int treecmp(struct node *tree1, struct node *tree2)
{
   coproc_t walk1, walk2;
   char equal = 1, continu = 1;
   char *val1, *val2;

   walk1 = costart(&treewalk,NULL);     /* Start a coprocess for   */
   walk2 = costart(&treewalk,NULL);     /* each tree.              */

   cocall(walk1, (char *) tree1);       /* Tell them what to walk. */
   cocall(walk2, (char *) tree2);

   while(continu) {

         /* Get a node from each tree.                            */
      val1 = cocall(walk1, &continu);
      val2 = cocall(walk2, &continu);

         /* See if either tree has run out of data.               */
      continu = val1 && val2;

         /* Compare the nodes.                                    */
      equal = (val1? strcmp(val1, val2) == 0: val2 == NULL);

         /* Leave loop on inequality.                             */
      if (!equal) continu = 0;
   }

      /* Terminate unfinished coprocesses.                        */
   if (val1)
      cocall(walk1, &continu);
   if (val2)
      cocall(walk2, &continu);

      /* Return the result.                                       */
   return equal;
}
```

* This example was adapted from S. Krogdahl and K. A. Olsen. "Ada, As Seen From Simula." *Software - Practice and Experience* 16, no. 8 (August 1986).

```
char *treewalk(char *head)
{
   struct node *tree;

   tree = (struct node *) head;         /* Get tree to walk.       */
   coreturn(NULL);                      /* Say that we're ready.   */
   walk(tree);                          /* Start walking.          */
   return NULL;                         /* Return 0 when done.     */
}

void walk(struct node *tree)
{
   if (tree->before)
      walk(tree->before);               /* Walk left subtree.      */
   if (!*coreturn(tree->value))         /* Return value, check for */
      coexit(NULL);                     /* termination.            */
   if (tree->after)
      walk(tree->after);                /* Walk right subtree.     */
}
```

Advanced Topics: Program Termination

When a program that uses coprocesses terminates, either by a call to **exit** or by a call to **return** from the main function, all coprocesses must be terminated. The process by which termination occurs is a complicated one whose details may be relevant to some applications.

By using the **blkjmp** function, you can intercept coprocess termination when it occurs as the result of **coexit** or **exit**, allowing the coprocess to perform any necessary cleanup. Both **coexit** and **exit** are implemented by the library as a **longjmp** to a jump buffer defined by the library; if you intercept an **exit** or **coexit**, you can allow coprocess termination to continue by issuing **longjmp** using the data stored in the jump buffer by **blkjmp**. (See the **blkjmp** function description in Chapter 8, "Program Control Functions," in *SAS/C Library Reference, Volume 1* for details of this process.)

The **atcoexit** function also can be used to establish a cleanup routine for a coprocess. Note that this function allows one coprocess to define a cleanup routine for another or even for every terminating coprocess.

Exit Processing for Secondary Coprocesses

When **exit** is called by a coprocess other than the main one, the library effects a **coexit** for each non-idle coprocess. More specifically, the following steps are performed:

1. A **longjmp** is performed using a library jump buffer to terminate the calling coprocess, allowing the termination to be intercepted by **blkjmp**. This means that the **exit** call is treated at first as a **coexit**.
2. Any **atcoexit** routines for the coprocess are called.
3. The active coprocess is terminated by the library. The coprocess that cocalled the terminated process is then made active. Instead of resuming the **cocall** function, the **exit** function is called again. If this coprocess is not the main coprocess, steps 1 and 2 are performed for that coprocess. If this coprocess is the main coprocess, the normal function of exit is performed and the entire program is terminated, as described in "Exit Processing for the Main Coprocess" on page 9-10.

Note that as a result of the above approach, by the time **exit** is called by the main coprocess, all remaining coprocesses are idle.

Exit Processing for the Main Coprocess

When the main coprocess calls **exit**, the following steps are performed:

1. A **longjmp** is performed using a library jump buffer to terminate the program, allowing termination to be intercepted by **blkjmp**.
2. Any **coexit** routines and any **atcoexit** routines for the main coprocess are called.
3. The main coprocess becomes ended.
4. If there are any idle coprocesses left, one of them is selected for termination. This coprocess is suspended in a call to the **coreturn** function. This coprocess becomes active, but the call to **coreturn** is not completed. Instead, a **coexit** is performed. (If, on the other hand, all coprocesses are ended, the remainder of program termination processing is performed.)
5. As usual, if the coprocess has issued **blkjmp**, it will intercept the **coexit** and can perform cleanup processing. Note that since there is no calling process to return to, an attempt to call **coreturn** is treated as an error. It is permissible to call **cocall** to communicate with other unterminated coprocesses or even to use **costart** to create new ones at this time.
6. When **coexit** completes, the current coprocess is terminated, and control is transferred to step 3.

All coprocesses are terminated before files are closed by the library so that coprocess termination routines can flush buffers or write final messages.

Advanced Topics: Signal Handling

To understand this section, you should be familiar with the normal operation of signal handling. You may want to review Chapter 5, "Signal-Handling Functions," of *SAS/C Library Reference, Volume 1* before proceeding.

Signal handling for programs that use coprocesses is complex because program requirements differ depending on the situation and the signals involved. For instance, for computational signals such as **SIGFPE**, you may prefer a separate signal handler for each coprocess. On the other hand, for a signal such as **SIGINT** that can occur at any time, independent of coprocess switches, you may prefer a single handler, which is called regardless of the coprocess active at the time of the signal. The coprocess implementation enables you to define local and global handlers for each signal independently in whatever way is most convenient for your application.

In addition, the library maintains a separate signal-blocking mask for each coprocess. This allows coprocesses to block signals that they are not equipped to handle, while assuring that signals are recognized whenever an appropriate coprocess becomes active. By default, all coprocesses except the main coprocess are created with all signals blocked. For this reason, unless a new coprocess issues **sigsetmask** or **sigprocmask**, no asynchronous signals are detected during its execution. Therefore, you can write subroutines that create coprocesses, and you do not need any special code to avoid interference with the asynchronous signal handling of the rest of the program. Note that you can use the second argument to **costart** to specify a different signal mask for the new coprocess.

When you use coprocesses, there are two functions available to define signal handlers: **signal** and **cosignal**. **signal** defines a *local handler*, a function that is called only for signals discovered while the coprocess that defined the handler is active. **cosignal** defines a *global handler*, a function that is called for signals discovered during the execution of any coprocess. If, at the time of signal discovery,

both a local and global handler are defined, the local handler is called unless the local handler is **SIG_DFL**, in which case the global handler is called.

Usually, you want to define local handlers (using the **signal** function) for synchronous signals and global handlers (using the **cosignal** function) for asynchronous signals. For instance, a **SIGFPE** handler normally is defined with **signal** because **SIGFPE** cannot occur while a coprocess is active except as the result of code executed by that coprocess. On the other hand, a **SIGINT** handler normally is defined with **cosignal** because the time at which a **SIGINT** signal is generated has no relationship to coprocess activity.

The local and global signal handlers are maintained independently. A call to **signal** returns the previous local handler for the calling coprocess; a call to **cosignal** returns the previous global handler.

You can also use **sigaction** to establish local or global handlers. Ordinarily, **sigaction** defines a local handler. You can set the **sa_flags** bit **SA_GLOBAL** to define a global handler instead.

Note that when a signal handler is called, no coprocess switch takes place; the handler executes as part of the interrupted coprocess. If you want a signal to be handled by a particular coprocess, you either can have all other coprocesses block the signal or you can define a global handler that cocalls the relevant coprocess, if that coprocess was not the one interrupted. An example of the latter technique is shown in the **cosignal** function description.

Note that global handlers generally should not use **longjmp** unless they first verify that the active coprocess is the one that issued the corresponding call to **setjmp**.

Function Descriptions

Descriptions of each coprocessing function follow. 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.

atcoexit Register Coprocess Cleanup Function

SYNOPSIS

```
#include <coproc.h>

int atcoexit(void (*func)(void), coproc_t procid);
```

DESCRIPTION

The **atcoexit** function defines a function called during termination of a particular coprocess or of all coprocesses either as the result of a call to **coexit** or a return from the coprocess' initial function. The **func** argument should be a function with no arguments returning **void**. The **procid** argument specifies the ID of the coprocess whose termination is to be intercepted or **0** to intercept termination of all coprocesses.

atcoexit routines free resources associated with particular coprocesses. These can be library-managed resources, such as load modules or FILEs, because these normally are cleaned up only when the entire program terminates. **atcoexit** can be called any number of times, and the same routine can be registered more than once, in which case it is called once for each registration.

atcoexit cleanup routines are called in the opposite order of their registration, and they execute as part of the terminating coprocess. They are called after termination of the coprocess' active functions. (Thus, a cleanup routine cannot cause coprocess execution to resume by issuing **longjmp**.) A cleanup routine can call **coexit**, which has no effect other than possibly changing the information returned to the cocalling coprocess. In this case, no cleanup routine previously called is called again during termination of the same coprocess.

It is not possible to deregister a function once registered. However, when a load module containing a registered cleanup routine is unloaded using **unloadm**, the cleanup routine is deregistered automatically.

You can call **atcoexit** during the termination of a coprocess, but the corresponding function is not called during termination of this coprocess. (Normally, it is called during the termination of other coprocesses to which it applies.)

Note that **atexit(func)** is equivalent to **atcoexit(func, coproc(MAIN))**.

RETURN VALUE

atcoexit returns **0** if successful or a nonzero value if unsuccessful.

PORTABILITY

atcoexit is not portable.

EXAMPLE

This example defines a routine **coalloc** that can be called to allocate memory belonging to a particular coprocess. An **atcoexit** cleanup routine is defined to free the memory allocated by each coprocess when it terminates.

```
#include <stdlib.h>
#include <coproc.h>
```

atcoexit Register Coprocess Cleanup Function
(continued)

```
struct memelt {    /* header for each block allocated  */
   struct memelt *fwd;
   coproc_t owner;
   double pad [0]; /* force correct alignment          */
};

static int first = 1;
static struct memelt *queue;
static void cleanup(void);

void *coalloc(int amt)
{
   struct memelt *newmem;

   if (first){
      if (atcoexit(cleanup, 0))
         abort();
      else
         first = 0;
   }

   newmem = (struct memelt *) malloc(amt + sizeof(struct memelt));
   if (!newmem) return 0;
   newmem->owner = coproc(SELF);
   newmem->fwd = queue;
   queue = newmem;
   return ((char *) newmem) + sizeof(struct memelt);
}

void cleanup()
{
   coproc_t ending = coproc(SELF);
   struct memelt **prev, *cur;

   for (prev = &queue; cur = *prev;) {
      if (cur->owner == ending) {
         *prev = cur->fwd;
         free(cur);
      }
      else
         prev = &cur->fwd;
   }
}
```

RELATED FUNCTIONS

blkjmp, **atexit**

cocall Pass Control to a Coprocess

SYNOPSIS

```
#include <coproc.h>

void *cocall(coproc_t procid, void *arg);
```

DESCRIPTION

The **cocall** function gives control to a coprocess and passes it a pointer value. Execution of the calling coprocess is suspended until the called coprocess calls **coreturn** or is terminated. The **procid** argument identifies the coprocess to be called. The **arg** value is a value passed to the called coprocess. **arg** must not have the value **CO_ERR**.

The **arg** value is passed to the called coprocess as follows:

- If this coprocess has not been called previously, its initial function (as established when the coprocess was costarted) is invoked with the specified argument value. For example, the following sequence executes the function call **(*f)(arg)** that begins executing the coprocess created by the **costart** call:

```
id = costart(f,NULL);
cocall(id,arg);
```

- If the called coprocess has been called previously, it is necessarily suspended as a result of the execution of a call to the **coreturn** function. In this case, the suspended call to **coreturn** is resumed, and **coreturn** returns the value specified by **arg**.

RETURN VALUE

cocall returns a pointer specified by the called coprocess. Depending on the circumstances, any of the following can occur. If the called coprocess suspends itself by calling **coreturn(val)**, the value **val** is returned by **cocall**. If the called coprocess terminates by calling **coexit(val)**, the value **val** is returned by **cocall**. If the called coprocess terminates as a result of execution of **return val;** by its initial function, the value **val** is returned by **cocall**.

If the **procid** argument is invalid (that is, if it identifies an invalid or terminated coprocess), the calling coprocess is not suspended, and **cocall** immediately returns the constant **CO_ERR**.

CAUTIONS

Recursive **cocall**s are not allowed. While a coprocess is suspended as the result of a call to **cocall**, it cannot be cocalled again. In particular, the main coprocess can never be cocalled. Any attempt to perform a recursive cocall returns **CO_ERR**.

cocall Pass Control to a Coprocess
(continued)

EXAMPLE

```
#include <stddef.h>
#include <coproc.h>
#include <stdlib.h>

coproc_t inp_proc;           /* input process process ID          */

void *input(void *);

char *file_name = "ddn:input";
char *input_ln;           /* a line of input, returned by inp_proc */

   /* Create and call a coprocess that reads a line of data from a */
   /* file. The "input" function, which is the initial function of */
   /* the new coprocess, is given in the EXAMPLE for coreturn.     */

inp_proc = costart(&input,NULL);          /* Create the coprocess. */

   /* Pass the input file name and check for error.               */
if (!cocall(inp_proc, file_name))
   exit(16);

for (;;) {
   input_ln = cocall(inp_proc, NULL);

         /* Ask for an input line; stop if no data left.          */
   if (!input_ln) break;
   .
   .   /* Process input data.                                     */
   .
}
```

coexit Terminate Coprocess Execution

SYNOPSIS

```
#include <coproc.h>

void coexit(void *val);
```

DESCRIPTION

coexit terminates execution of the current coprocess and returns a pointer value to its cocaller. The call to **cocall** that called the terminating coprocess is resumed, and it returns the value specified by **val**.

If **coexit** is called by the main coprocess, it is treated as a call to exit. Specifically, **coexit(val)** is treated as **exit(val? *(int *) val: 0)**.

coexit is implemented as a **longjmp** to a location within the library coprocess initialization routine. This means that the **blkjmp** function can be used within a coprocess to intercept calls to **coexit**.

If program execution is terminated by a call to the **exit** function while there are still coprocesses active, each active coprocess is terminated by an implicit call to **coexit**. These calls also can be intercepted by **blkjmp** ; however, note that in this context any use of **coreturn** fails, because no calling coprocess is defined.

RETURN VALUE

Control does not return from **coexit**.

EXAMPLE

See the example for **coreturn**.

RELATED FUNCTIONS

blkjmp, **exit**

coproc Return the ID of a Coprocess

SYNOPSIS

```
#include <coproc.h>

coproc_t coproc(int name);
```

DESCRIPTION

coproc returns the coprocess identifier of a particular coprocess as specified by its integer argument. The argument should be specified as one of these three symbolic values: **MAIN**, **SELF**, or **CALLER**.

coproc(MAIN) returns the coprocess ID of the main coprocess, which is always created during program initialization. **coproc(SELF)** returns the ID of the coprocess currently running. **coproc(CALLER)** returns the ID of the coprocess that cocalled the current coprocess.

RETURN VALUE

coproc returns the ID of the specified coprocess or `0` if the argument does not specify a valid coprocess. **coproc(CALLER)** returns `0` if called from the main coprocess. Also note that **coproc(MAIN)** returns `0` if called after a call to **exit**, causing program termination.

EXAMPLE

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

   /* Test whether the main coprocess or some */
   /* other coprocess is running.             */
if (coproc(SELF) == coproc(MAIN))
   printf("main coprocess currently active.\n");
```

coreturn Return Control to a Coprocess

SYNOPSIS

```
#include <coproc.h>

void *coreturn(void *arg);
```

DESCRIPTION

coreturn returns control to the current cocaller of a coprocess and returns a pointer value. Execution of the returning coprocess is suspended until it is cocalled again. The **arg** value is a value to be returned to the cocaller of the current coprocess. **arg** must not have the value **CO_ERR**.

The **arg** value is passed to the resumed coprocess as follows. By necessity, the caller of the current process is suspended by executing the **cocall** function. This call is resumed and returns the value specified by **arg** to its caller.

RETURN VALUE

coreturn returns only when the current coprocess is cocalled again by some other coprocess. The return value is the **arg** value specified by the corresponding call to **cocall**.

An invalid call to **coreturn** immediately returns the value **CO_ERR**.

CAUTION

coreturn cannot be called from the main coprocess because it has no cocaller to return to.

EXAMPLE

This example is a companion to the **cocall** example. It illustrates a coprocess whose purpose is to read a file and coreturn a line at a time.

```
#include <stddef.h>
#include <coproc.h>
#include <stdio.h>

void *input(void *filename)
{
   FILE *f;
   char buf [100];
   f = fopen(filename, "r");

       /* if open failed, quit, pass NULL to calling coprocess     */
   if (!f)
      coexit(NULL);

      /* else acknowledge valid name                               */
   coreturn(filename);
```

coreturn Return Control to a Coprocess
(continued)

```
   for (;;) {
      fgets(buf, 100, f);            /* Read the next line.        */
      if (feof(f)) {                /* if reached end-of-file      */
         fclose(f);
         coexit(NULL);  /* Terminate coprocess and return NULL.    */
      }
            /* else pass back input line address                   */
      else coreturn(buf);
   }
}
```

cosignal Define a Global Signal Handler

SYNOPSIS

```
#include <coproc.h>
#include <signal.h>

   /* This typedef is in <coproc.h>. */
typedef __remote void(*_COHANDLER)(int);
_COHANDLER cosignal(int signum, _COHANDLER handler);
```

DESCRIPTION

The **cosignal** function defines a global handler for a signal. A global handler can be called during the execution of any coprocess except one that has used the **signal** or **sigaction** function to define a local handler for the same signal. The **signum** argument is the number of the signal, which should be specified as a symbolic signal name.

The **handler** argument specifies the function to be called when the signal occurs. The **handler** argument can be specified as one of the symbolic values **SIG_IGN** or **SIG_DFL** to specify that the signal should be ignored or that the default action should be taken, respectively.

The handler always executes as part of the coprocess interrupted by the signal.

RETURN VALUE

cosignal returns the address of the handler for the signal established by the previous call to **cosignal** or **SIG_DFL** if **cosignal** has not been previously called. **cosignal** returns the special value **SIG_ERR** if the request cannot be honored.

CAUTIONS

Use of **cosignal** has no effect on the handling of a signal that is discovered during execution of a coprocess that has defined a local handler (other than **SIG_DFL**) using the **signal** function. However, when **SIG_DFL** is defined as the local handler, any global handler will be called.

Handlers established with **cosignal** should not call **longjmp** without verifying that the coprocess executing is the one that called **setjmp** to define the target. Attempting to use **longjmp** to transfer control in one coprocess to a target established by another causes the program to terminate abnormally.

cosignal is more appropriate than **signal** for handling asynchronous signals in a multiple coprocess program because asynchronous signals can occur at any time, regardless of transfers of control between processes. Alternately, because each coprocess has its own mask of blocked signals, you can use **sigblock** and **sigsetmask** to prevent signals from being discovered except during the execution of a process set up to handle them.

EXAMPLE

The following example shows how a coprocess can be defined to get control every *n* seconds. The coprocess uses **cosignal** to define a global handler for

cosignal Define a Global Signal Handler
(continued)

the **SIGALRM** signal. This handler then uses **cocall** to give control to the coprocess.

```
#include <stddef.h>
#include <coproc.h>
#include <signal.h>
#include <lclib.h>

coproc_t tmr_proc;        /* timer coprocess ID                    */
char *tmr_wait(char *);
int delay = 10;           /* delay time in seconds                 */

main()
{
      /* Create the timer coprocess.                               */
   tmr_proc = costart(&tmr_wait,NULL);
      /* Allow timer coprocess to initialize.                      */
   cocall(tmr_proc, NULL);
      /* Set time until first signal.                              */
   alarm(delay);
   .
   .
   .
}

void tmr_hndl(int signum)
{
   /* This is the global SIGALRM handler, which uses cocall to     */
   /* activate the timer coprocess. After the timer coprocess has  */
   /* handled the signal, alarm is called to establish the next    */
   /* interval.                                                    */

   cocall(tmr_proc, NULL);
   alarm(delay);             /* Start new timer interval.          */
}

char *tmr_wait(char *unused)
{
   /* This function is a coprocess that establishes a global       */
   /* signal handler with cosignal to gain control every time      */
   /* SIGALRM is raised. The handler is responsible for calling    */
   /* alarm to re-establish the interval after handling is         */
   /* complete. Note that this technique assures that SIGALRM will */
   /* not be raised while this coprocess is active.                */
```

cosignal Define a Global Signal Handler
(continued)

```
        for (;;) {                          /* Do until program exit.    */
           cosignal(SIGALRM, &tmr_hndl);     /* Define global handler.   */
           coreturn(NULL);                   /* Let rest of program run. */
           .
           .           /* Alarm has gone off -- do periodic cleanup.    */
           .

        }
     }
```

RELATED FUNCTIONS

sigaction, **signal**

SEE ALSO

See Chapter 5, "Signal-Handling Functions," in the *SAS/C Library Reference, Volume 1.*

costart Create a New Coprocess

SYNOPSIS

```
#include <coproc.h>

coproc_t costart(void *(*func)(void *), struct costart_parms *p);
```

DESCRIPTION

The **costart** function creates a new coprocess. Its argument is the address of the initial function to be executed. The initial function should accept a **void *** argument and return a **void *** value. This function is not called until the first time the new coprocess is cocalled.

The **p** argument can be **NULL** or a pointer to a **struct costart_parms**, which has the following definition (in **<coproc.h>**):

```
struct costart_parms {
   unsigned stksize;
   unsigned sigmask;
   sigset_t *blocked_set;
   unsigned _[3];
};
```

If **p** is a non-**NULL** pointer, then the values of each of the members at the **struct costart_parms** it points to is used when the coprocess is created. If **p** is **NULL**, then default values are used.

The value in **stksize** is used as the size of the initial stack allocation for the coprocess. This value must be at least 680 bytes. All values are rounded up to an even fraction of 4096. The default initial stack allocation is 4096. Use of **stksize** does not prevent the stack from being extended automatically if a stack overflow occurs.

The **sigmask** and **blocked_set** arguments are used to specify the signal mask for the coprocess. The **sigmask** field can be used to mask signals that are managed by SAS/C, and the **blocked_set** argument can be used to mask signals that are managed by either SAS/C or OpenEdition. **sigmask** is provided primarily for backwards compatibility, and it is recommended that you use the **sigprocmask** function to establish a **sigset_t** object that contains the mask information for both SAS/C and OpenEdition managed signals. The **blocked_set** argument is then used to point to the **sigset_t** object.

If **blocked_set** is set to **0**, the value of **sigmask** is used to establish the signal mask for the coprocess. The default value of **sigmask** is **0xffffffff**; that is, all signals are blocked. This blocks interruptions until the new coprocess is ready. Refer to "Blocking Signals" in Chapter 5 of *SAS/C Library Reference, Volume 1* for more information about the signal mask.

The **_[3]** field is reserved and must be set to **0**s.

RETURN VALUE

costart returns the ID of the new coprocess or **0** if a new coprocess could not be created.

costart Create a New Coprocess
(continued)

CAUTIONS

All of the members in the **costart_parms** structure must be used. If you want to change only one of the fields, you must set the other fields to their default value.

At most, 65,536 coprocesses (including the main coprocess created at program start-up) can exist simultaneously. In practice, because each coprocess needs stack space beow the 16-megabyte line, it will not be possible to create more than roughly 10,000 coprocesses. The exact limit depends on the operating system and region or virtual machine size.

RELATED FUNCTIONS

sigaddset

SEE ALSO

See Chapter 5, "Signal-Handling Functions," in the *SAS/C Library Reference, Volume 1.*

costat Return Coprocess Status

SYNOPSIS

```
#include <coproc.h>

int costat(coproc_t id);
```

DESCRIPTION

The **costat** function returns information about the status of a coprocess. The **id** argument is the ID of the coprocess whose status is desired.

RETURN VALUE

The value returned by **costat** is one of the symbolic constants **STARTING**, **ACTIVE**, **BUSY**, **IDLE**, or **ENDED**. These constants have the following significance:

STARTING
indicates that the coprocess has been created by a call to **costart** but has never been cocalled.

ACTIVE
indicates that the coprocess is the one currently running.

BUSY
indicates that the coprocess has cocalled another coprocess and therefore has been suspended by the **cocall** function. A **BUSY** coprocess cannot be cocalled itself.

IDLE
indicates that the coprocess is suspended by a call to **coreturn**. The coprocess will not execute again until it is cocalled.

ENDED
indicates that the coprocess has terminated.

CAUTION

The effects of passing an invalid coprocess ID to **costat** are unpredictable. Usually, this causes the value **ENDED** to be returned.

costat Return Coprocess Status
(continued)

EXAMPLE

```
#include <stddef.h>
#include <coproc.h>
#include <stdlib.h>

coproc_t err_proc;

/* Cocall the coprocess whose ID is in the variable err_proc. But  */
/* abort instead if that coprocess cannot be legally cocalled.     */
switch (costat(err_proc)) {
   case STARTED:
   case IDLE:
      cocall(err_proc,NULL);       /* Call the error coprocess if  */
      break;                       /* this is legal.               */
   default:
      abort();                     /* Abort if error coprocess not */
}                                 /* available for cocall.        */
```

10 Localization

10-1 Introduction
10-1 Locales and Categories
10-2 The Locale Structure lconv
10-4 The "S370" Locale
10-4 The "POSIX" Locale
10-4 Library-Supplied Locales
10-5 Function Descriptions
10-17 User-Added Locales
10-17 Creating a New Locale

Introduction

The functions defined in the header file **<locale.h>** are used to tailor significant portions of the C run-time environment to national conventions for punctuation of decimal numbers, currency, and so on. This tailoring provides the capability for writing programs that are portable across many countries. For example, some countries separate the whole and fractional parts of a number with a comma instead of a period. Alphabetization, currency notation, dates, and times are all expressed differently in different countries. Each set of definitions for culture-dependent issues is called a *locale*, and each locale has a name that is a null-terminated string.

This chapter introduces fundamental concepts of localization and the basic structure used in localization functions, and discusses the three nonstandard locales supplied by the SAS/C Library. Descriptions of the standard localization functions (including **strcoll** and **strxfrm**) follow.

For details on how to create your own locales, see "User-Added Locales" on page 10-17 .

Locales and Categories

SAS/C defines the following locales:

''S370'' is a locale that defines conventions traditionally associated with the 370 mainframe environment.

''POSIX'' is a locale that defines conventions traditionally associated with UNIX implementations, as codified by the POSIX 1003.2 standard.

''C'' is the locale in effect when your program begins execution. For programs called with **exec** linkage, this is the same as the **''POSIX''** locale. For other programs, it is the same as the **''S370''** locale.

'''' is a locale that represents the best fit for local customs. The **''''** locale interpretation is controlled by several environment variable settings. See the **setlocale** function description for more information.

Three other locales are supplied by the SAS/C Library. For details on these locales, see "Library-Supplied Locales" on page 10-4 . As mentioned earlier, you can also supply your own locales; see "User-Added Locales" on page 10-17 for details.

A locale is divided into *categories*, which define different parts of the whole locale. You can change one category without having to change the entire locale. For example, you can change the way currency is displayed without changing the expression of dates and time. The categories of locale defined in **<locale.h>** are as follows:

LC_ALL	affects all the categories at once.
LC_COLLATE	affects the collating sequence. This changes how **strcoll** and **strxfrm** work.
LC_CTYPE	affects how the character type macros (such as **isgraph**) work. **LC_CTYPE** also affects the multibyte functions (such as **mblen** and **wcstombs**) as well as the treatment of multibyte characters by the formatted I/O functions (such as **printf** and **sprintf**) and the string functions. **isdigit** and **isxdigit** are not affected by **LC_CTYPE**.
LC_MONETARY	affects how currency values are formatted.
LC_NUMERIC	affects the character used for the decimal point.
LC_TIME	affects how **strftime** formats time values. This category does not affect the behavior of **asctime**.

The Locale Structure lconv

<locale.h> defines the structure **struct lconv**. The standard members of this structure are explained in the following list. The default **''C''** locale values, as defined by the ANSI and ISO C standards, are in parentheses after the descriptions. A default value of CHAR_MAX indicates the information is not available.

char *decimal_point
is the decimal point character used in nonmonetary values. (**''.''**)

char *thousands_sep
is the character that separates groups of digits to the left of the decimal point in nonmonetary values. (**''''**)

char *grouping
is the string whose elements indicate the size of each group of digits in nonmonetary values. (**''''**)

char *int_curr_symbol
is the international currency symbol that applies to the current locale. The fourth character, immediately before the null character, is the character used to separate the international currency symbol from the value. (**''''**)

char *currency_symbol
is the local currency symbol that applies to the current locale. (**''''**)

char *mon_decimal_point
is the decimal point used in monetary values. (**''''**)

char *mon_thousands_sep
is the character that separates groups of digits to the left of the decimal point in monetary values. (**''''**)

char *mon_grouping
is the string whose elements indicate the size of each group of digits in monetary values. (**''''**)

char *positive_sign
is the string that indicates a nonnegative monetary value. (**''''**)

char *negative_sign
is the string that indicates a negative monetary value. (**''**)

char int_frac_digits
is the number of fractional digits to be displayed in an internationally-formatted monetary value. (**CHAR_MAX**)

char frac_digits
is the number of fractional digits to be displayed in a locally-formatted monetary value. (**CHAR_MAX**)

char p_cs_precedes
is set to **1** if the **currency_symbol** and a nonnegative monetary value are separated by a space, otherwise it is set to **0**. (**CHAR_MAX**)

char p_sep_by_space
is set to **1** if the **currency_symbol** comes before a nonnegative monetary value; it is set to **0** if the **currency_symbol** comes after the value. (**CHAR_MAX**)

char n_cs_precedes
is set to **1** if the **currency_symbol** and a negative monetary value are separated by a space, otherwise it is set to **0**. (**CHAR_MAX**)

char n_sep_by_space
is set to **1** if the **currency_symbol** comes before a negative monetary value; it is set to **0** if the **currency_symbol** comes after the value. (**CHAR_MAX**)

char p_sign_posn
is set to a value indicating the position of the **positive_sign** for a nonnegative monetary value. (**CHAR_MAX**)

char n_sign_posn
is set to a value indicating the position of the **negative_sign** for a negative monetary value. (**CHAR_MAX**)

The elements of **grouping** and **mon_grouping** are defined by the following values:

CHAR_MAX	indicates no more grouping is to be done. **CHAR_MAX** is defined in **<limits.h>** as **255**.
0	indicates that the previous element is to be repeatedly used for the rest of the digits.
other	is the number of digits that make up the current group. The next element is examined to determine the size of the next group to the left.

The value of **p_sign_posn** and **n_sign_posn** is defined by the following:

0 indicates that parentheses surround the value and **currency_symbol**.

1 indicates that the sign comes before the value and **currency_symbol**.

2 indicates that the sign comes after the value and **currency_symbol**.

3 indicates that the sign comes immediately before the **currency_symbol**.

4 indicates that the sign comes immediately after the **currency_symbol**.

The “S370” Locale

The **``S370''** locale defines its categories as follows:

LC_COLLATE causes **strxfrm** to behave like **strncpy**, except that it returns the number of characters copied, not a pointer to a string.

LC_CTYPE The 1403 PN print train is used as a reference to determine whether characters are considered printable.

LC_MONETARY has no effect in this locale.

LC_NUMERIC has no effect in this locale.

LC_TIME causes **strftime** to return the same values for days of the week, dates, and so on, as **asctime**.

The “POSIX” Locale

The **``POSIX''** locale defines its categories as follows:

LC_COLLATE causes comparisons to be performed according to the ASCII collating sequence rather than EBCDIC.

LC_CTYPE specifies that characters are considered to be printable if they are defined as printable in ASCII by the ISO C Standard.

LC_MONETARY has no effect in this locale.

LC_NUMERIC has no effect in this locale.

LC_TIME causes **strftime** to return the same values for days of the week, dates, and so on, as **asctime**.

Library-Supplied Locales

Besides the built-in **``S370''** locales, the run-time library supplies three other locales in source form or load form or both. The load module names can be found in L$CL*xxxx* where *xxxx* is the name of the locale. See your SAS Software Representative for SAS/C sofware products for the location of the source code.

``DBCS'' is a double-byte character set locale and is supplied as a run-time load module. It allows recognition and processing of double-byte character strings by various library functions when specified for category **LC_CTYPE** or **LC_ALL**.

For category **LC_COLLATE**, this locale enables **strxfrm** and **strcoll** to transform and collate mixed double-byte character strings. In all other aspects, the **``DBCS''** locale is identical to the **``S370''** locale.

``SAMP'' is a sample locale that is distributed in source form and has the following category characteristics:

LC_CTYPE
is a character type table (ctype) identical to the **``S370''** locale.

LC_COLLATE
is a single-byte collation table identical to the **``S370''** locale.

LC_NUMERIC
uses United States (USA) conventions.

LC_MONETARY
uses United States (USA) conventions.

LC_TIME
uses **''C''** locale conventions with a modified sample date routine used with **strftime %x** format.

''DBEX'' is a "double-byte" example locale that is distributed in source form and has the following characteristics:

LC_CTYPE
allows DBCS string recognition. It uses the default "S370" table since the **ctype** table pointer is **NULL**.

LC_COLLATE
is a double-byte collation table provided for altering the DBCS collating sequence as well as sample **strxfrm** and **strcoll** functions that use the table.

LC_NUMERIC
is unavailable. (The **''S370''** locale is used.)

LC_MONETARY
is unavailable. (The **''S370''** locale is used.)

LC_TIME
is unavailable. (The **''S370''** locale is used.)

Function Descriptions

Descriptions of each localization function follow. 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. None of the localization functions is supported by traditional UNIX C compilers.

localeconv Get Numeric Formatting Convention Information

SYNOPSIS

```
#include <locale.h>

struct lconv *localeconv(void);
```

DESCRIPTION

localeconv sets the elements of an object of type **struct lconv** to the appropriate values for the formatting of numeric objects, both monetary and nonmonetary, with respect to the current locale.

RETURN VALUE

localeconv returns a pointer to a filled-in object of type **struct lconv**. The **char *** members of this structure may point to **''**, indicating that the value is not available or is of **0** length. The **char** members are nonnegative numbers and can be equal to **CHAR_MAX**, indicating the value is not available. See "The Locale Structure lconv" on page 10-2 for a discussion of each structure member returned by **localeconv**.

CAUTIONS

Another call to **localeconv** overwrites the previous value of the structure; if you need to reuse the previous value, be sure to save it. The following code saves the value of the structure:

```
struct lconv save_lconv;
save_lconv = *(localeconv());
```

Also, calls to **setlocale** with categories **LC_ALL**, **LC_MONETARY**, or **LC_NUMERIC** may overwrite the contents of the structure returned by **localeconv**.

EXAMPLE

The following example converts integer values to the format of a locale-dependent decimal picture string. **frac_digits** indicates how many digits are to the right of the decimal point character. If **frac_digits** is negative, the number is padded on the right, before the decimal point character, with the same number of 0s as the absolute value of **frac_digits**. It is assumed that the digit grouping string has at most only one element.

```
#include <stdio.h>
#include <string.h>
#include <locale.h>

size_t dec_fmt();
```

localeconv Get Numeric Formatting Convention Information

(continued)

```
int main()

{
   struct lconv *lc;                /* pointer to locale values */
   char buf[256];

   setlocale(LC_NUMERIC, "SAMP");   /* Use "SAMP" locale.       */

      /* Call localeconv to obtain locale conventions.          */
   lc = localeconv();

   dec_fmt(buf, lc, 12345678, 0);
   printf("The number is: 12,345,678. == %s\n", buf);

   dec_fmt(buf, lc, 12345678, 2);
   printf("The number is: 123,456.78 == %s\n", buf);

   dec_fmt(buf, lc, -12345678, 4);
   printf("The number is: -1,234.5678 == %s\n", buf);

   dec_fmt(buf, lc, 12345678, -5);
   printf("The number is: 1,234,567,800,000. == %s\n", buf);

   dec_fmt(buf, lc, 12345678,10);
   printf("The number is: 0.0012345678 == %s\n", buf);
}

size_t dec_fmt(char *buf, struct lconv *lc, int amt,
               int frac_digits)
{
   char numstr[128];        /* number string                    */
   char *ns_ptr,            /* number string pointer            */
      *buf_start;           /* output buffer start pointer      */
   int ngrp,                /* number of digits per group       */
      ntocpy,               /* digits to copy                   */
      non_frac;             /* number of nonfractional digits   */
   size_t ns_len;           /* number string length             */


   if (abs(frac_digits) > 100) /* Return error if too big       */
      return 0;
   buf_start = buf;

   sprintf(numstr, "%+-d", amt); /* Get amount as number string */
   ns_ptr = numstr;             /* Point to number string      */
   ns_len = strlen(ns_ptr);     /* Get number string length    */
```

localeconv Get Numeric Formatting Convention Information

(continued)

```
   if (frac_digits < 0) {   /* zero pad left of decimal point    */
      memset(ns_ptr + ns_len, '0', (size_t) -frac_digits);
      *(ns_ptr + ns_len - frac_digits) = '\0';
      ns_len -= frac_digits;      /* Add extra digit length.     */
      frac_digits = 0;
   }

      /* zero pad right of decimal point                          */
   if ((non_frac = ns_len - frac_digits - 1) < 0) {
      sprintf(numstr,"%+0*d", ns_len - non_frac + 1, amt);
      non_frac = 1;               /* e.g., 0.000012345678         */
   }

   if (amt < 0) *buf++ = *ns_ptr;  /* Insert sign in buffer.       */
      ns_ptr++;                    /* Skip +/- in number string.   */

       /* Convert grouping to int.                                 */
   if (!(ngrp = (int) *(lc->grouping)))
      ngrp = non_frac;             /* Use non_frac len if none.    */

      /* Get number of digits to copy for first group.            */
   if (!(ntocpy = non_frac % ngrp))
   ntocpy = ngrp;

   while (non_frac > 0) {         /* Separate groups of digits.   */
      memcpy(buf, ns_ptr, ntocpy); /* Copy digits.                */
      ns_ptr += ntocpy;            /* Advance pointers.           */
      buf    += ntocpy;
      if (non_frac -= ntocpy) {       /* Insert separator and set */
         *buf++ = *(lc->thousands_sep);/* number of digits for other */
         ntocpy = ngrp;               /* groups.                  */
      }
   }

   *buf++ = *(lc->decimal_point);      /* Insert decimal point.    */

   if (frac_digits > 0)
         /* Copy fraction + '\0' */
      memcpy(buf, ns_ptr, frac_digits + 1);
   else *buf = '\0';                  /* Else just null-terminate. */
   return strlen(buf_start);          /* Return converted length. */

}
```

setlocale Select Current Locale

SYNOPSIS

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

DESCRIPTION

setlocale selects the current locale or a portion thereof, as specified by the **category** and **locale** arguments. Here are the valid categories:

LC_ALL	names the overall locale.
LC_COLLATE	affects the behavior of **strcoll** and **strxfrm**.
LC_CTYPE	affects the behavior of the character type macros and the multibyte functions.
LC_MONETARY	affects the monetary-value formatting information returned by **localeconv**.
LC_NUMERIC	affects the decimal point character used by the I/O and string conversion functions, and the nonmonetary formatting information returned by **localeconv**.
LC_TIME	affects the behavior of **strftime**.

locale points to a **locale_string** that has one of the following formats:

- ***xxxx***

  Here is an example:

  ```
  setlocale(LC_ALL, "C");
  ```

- ***xxxx;category=yyyy<;category=zzzz ...>***

  Here is an example:

  ```
  setlocale(LC_MONETARY, "C;LC_TIME=SAMP");
  ```

- ***category=yyyy<;category=zzzz ...>***

  Here is an example:

  ```
  setlocale(LC_MONETARY, "LC_TIME=SAMP");
  ```

- **''** (null string)
- **NULL** (pointer)

  Here is an example:

  ```
  setlocale(LC_ALL, NULL);
  ```

setlocale Select Current Locale *(continued)*

In the **locale_string** formats, *xxxx*, *yyyy*, and *zzzz* have the following meanings:

- *xxxx* specifies the name for the requested category.
- *yyyy* and *zzzz* are overrides for mixed categories.
- *x*, *y*, and *z* can be uppercase alphabetic (A through Z), numeric (0 through 9), $, @, or #.

Any number of category overrides can be specified; however, only the first one per category is honored. Category overrides are honored only when the **LC_ALL** category is requested or the specific category and the override match. For example, this statement sets only the **LC_MONETARY** category; the **LC_TIME** override is ignored.

```
setlocale(LC_MONETARY,"C;LC_TIME=SAMP");
```

When **NULL** is specified, **setlocale** returns the current locale string in the same format as described earlier.

If the locale string is specified as **''''**, the library consults a number of environment variables to determine the appropriate settings. If none of the environment variables are defined, the **''C''** locale is used.

The locale **''''** is resolved in the following steps:

1. If the environment variable **LC_ALL** is defined and not NULL, the value of **LC_ALL** is used as the locale.
2. If a category other than **LC_ALL** was specified, and there is an environment variable with the same name as the category, the value of that variable is used (if not NULL). For instance, if the category was **LC_COLLATE**, and the environment variable **LC_COLLATE** is **''DBEX''**, then the **LC_COLLATE** component of the **''DBEX''** locale is used. Note that if the category is **LC_ALL**, the effect is the same as if each other category were set independently.
3. If the environment variable **LANG** is defined and not NULL, the value of **LANG** is used as the locale.
4. If the environment variable **_LOCALE** is defined and not NULL, the value of **_LOCALE** is used as the locale.

The first three steps are defined by the POSIX 1003.1 standard; the fourth step is for compatability with previous releases of SAS/C.

RETURN VALUE

setlocale returns the string associated with the specified **category** value. If the value for **locale** is not valid or an error occurs when loading the requested locale, **setlocale** returns **NULL** and the locale is not affected.

If **locale** is specified as **NULL**, **setlocale** returns the string associated with the **category** for the current locale and the locale is not affected.

When issuing **setlocale** for a mixed **locale**, if a category override fails (for example, if the locale load module cannot be found), then for that category the **LC_ALL** category is used if it is being set at the same time. Otherwise, a **NULL** return is issued and the category for which the locale override was requested remains unchanged. The returned string indicates the actual mixed locale in effect.

setlocale Select Current Locale
(continued)

CAUTIONS

A second call to **setlocale** overwrites the previous value, so you must save the current value before calling **setlocale** again, if you need the value later. This involves determining the string length for the locale name, allocating storage space for it, and copying it. The following code does all three:

```
char *name;
name = setlocale(LC_ALL, NULL);
name = strsave(name);
```

When you want to revert to the locale saved in **name**, use the following code:

```
setlocale(LC_ALL, name);
```

IMPLEMENTATION

Except for the **''S370''** locale, which is built-in, locale information is kept in a load module created by compiling and linking **''S370''** source code for the locale's localization data and routines. The load module name is created by appending up to the first four uppercased characters of the locale's name to the **''L$CL''** prefix. For example, if the locale name is **MYLOCALE**, the load module name is **L$CLMYLO**.

The locale load module is loaded when needed (with **loadm**) during **setlocale** processing. For the exact details and requirements on specifying localization data and routines, see "User-Added Locales" on page 10-17 . Also "loadm" on page 1-10 for load module library search and location requirements.

Mixed locale strings can be specified. In this case, all requested locales are loaded by a single call to **setlocale**.

EXAMPLE

```
#include <locale.h>

main()
{
char *name;

   /* Set all portions of the current locale to      */
   /* the values defined by the built-in "C" locale. */
setlocale(LC_ALL, "C");

   /* Set the LC_COLLATE category to the value       */
   /* defined by the "DBCS" locale.                  */
setlocale(LC_COLLATE, "DBCS");
```

setlocale Select Current Locale
(continued)

```
   /* Set the LC_CTYPE category to the value        */
   /* defined by the "DBCS" locale.                 */
setlocale(LC_CTYPE, "DBCS");

   /* Set the LC_NUMERIC category to the value      */
   /* defined by the "SAMP" locale.                 */
setlocale(LC_NUMERIC, "SAMP");

   /* Return mixed locale string.                   */
name = setlocale(LC_ALL, NULL);
}
```

The following string is pointed to by **name**:

```
"C;LC_COLLATE=DBCS;LC_CTYPE=DBCS;LC_NUMERIC=SAMP"
```

This string is interpreted as if the "**C**" locale (**LC_ALL**) is in effect except for **LC_COLLATE** and **LC_CTYPE**, which have the "**DBCS**" locale in effect, and **LC_NUMERIC**, which is using "**SAMP**".

The same results can be accomplished with a single call to **setlocale**:

```
#include <locale.h>

void main()
{
char *name;

name = setlocale(LC_ALL,
       "C;LC_COLLATE=DBCS;LC_CTYPE=DBCS;LC_NUMERIC=SAMP");
}
```

strcoll Compare Two Character Strings Using Collating Sequence

SYNOPSIS

```
#include <string.h>

int strcoll(const char *str1, const char *str2);
```

DESCRIPTION

strcoll compares two character strings (**str1** and **str2**) using the collating sequence or routines (or both) defined by the **LC_COLLATE** category of the current locale. The return value has the same relationship to **0** as **str1** has to **str2**. If two strings are equal up to the point where one of them terminates (that is, contains a null character), the longer string is considered greater.

Note that when the **''POSIX''** locale is in effect, either explicitly or by default, **strcoll** compares the strings according to the ASCII collating order rather than the EBCDIC order.

RETURN VALUE

The return value from **strcoll** is one of the following:

0	is returned if the two strings are equal.
less than **0**	is returned if **str1** compares less than **str2**.
greater than **0**	is returned if **str1** compares greater than **str2**.

No other assumptions should be made about the value returned by **strcoll**. In the "**S370**" locale, the **strcoll** return value is the same as if the **strcmp** function were used to compare the strings.

CAUTIONS

If one of the arguments of **strcoll** is not properly terminated, a protection or addressing exception may occur.

IMPLEMENTATION

strcoll uses the following logic when comparing two strings:

1. **strcoll** calls the locale's **strcoll** function equivalent, if available, and returns its value. See "LOCALE strcoll EQUIVALENT" on page 10-27.
2. **strcoll** calls the locale's **strxfrm** function equivalent, if available, to transform the strings for a character-by-character comparison. See "LOCALE strxfrm EQUIVALENT" on page 10-28.
3. **strcoll** calls the library's double-byte collation routine with a standard double-byte collating sequence if the locale is a double-byte locale as determined from the **LC_COLLATE** category, no locale **strcoll** or **strxfrm** function is available, and no collation table is supplied.
4. **strcoll** uses a collation table to compare the two strings (which are then compared character by character) if the locale is a single-byte locale and has a collation table available.
5. **strcoll** calls the **strcmp** function to compare the strings and returns its value if none of the above are true.

strcoll Compare Two Character Strings Using Collating Sequence
(continued)

EXAMPLE

```
#include <locale.h>
#include <string.h>
#include <stdio.h>

main()
{
   char *s1, *s2, *lcn;
   int result;

      /* Obtain locale name. */
   lcn = setlocale(LC_COLLATE, NULL);
   s1 = " A B C D";
   s2 = " A C B";

   result = strcoll(s1, s2);

   if (result == 0)
      printf("%s = %s in the \"%s\" locale", s1, s2, lcn);
   else
      if (result < 0)
         printf("%s < %s in the \"%s\" locale", s1, s2, lcn);
      else
         printf("%s > %s in the \"%s\" locale", s1, s2, lcn);
}
```

strxfrm Transform a String Using Locale-Dependent Information

SYNOPSIS

```
#include <string.h>

size_t strxfrm(char *str1, const char *str2, size_t n);
```

DESCRIPTION

strxfrm transforms the string pointed to by **str2** using the collating sequence or routines (or both) defined by the **LC_COLLATE** category of the current locale. The resulting string is placed into the array pointed to by **str1**.

The transformation is such that if the **strcmp** function is applied to two transformed strings, it returns greater than, equal to, or less than **0**, corresponding to the result of the **strcoll** function applied directly to the two original strings.

No more than **n** characters are placed into the resulting array pointed to by **str1**, including the terminating null character. If **n** is **0**, **str1** is permitted to be **NULL**. The results are unpredictable if the strings pointed to by **str1** and **str2** overlap.

RETURN VALUE

strxfrm returns the length of the transformed string (not including the terminating null character). If the value returned is **n** or more, the first **n** characters are written to the array without null termination. In the "**S370**" locale, the behavior of **strxfrm** is like that of **strncpy** except that the number of characters copied to the output array is returned instead of a pointer to the copied string.

CAUTION

If the **str2** argument is not properly terminated or **n** is bigger than the output array pointed to by **str1**, then a protection or addressing exception may occur. The size of the output array needed to hold the transformed string pointed to by **str2** can be determined by the following statement:

```
size_needed = 1 + strxfrm(NULL, str1, 0);
```

IMPLEMENTATION

strxfrm uses the following logic when transforming a string:

1. **strxfrm** calls the locale's **strxfrm** function equivalent, if available, to transform **str2** and place the result in the **str1** array. See "LOCALE strxfrm EQUIVALENT" on page 10-28.
2. **strxfrm** calls the library's double-byte **strxfrm** collation routine with a standard double-byte collating sequence if the locale is a double-byte locale as determined from the **LC_COLLATE** category, no locale **strxfrm** function is available, and no collation table is supplied.
3. **strxfrm** uses a collation table to transform the string if the locale is a single-byte locale and has a collation table available.
4. **strxfrm** invokes the equivalent of **strncpy** to copy **str2** to **str1** if none of the above are true.

strxfrm Transform a String Using Locale-Dependent Information
(continued)

EXAMPLE

This example verifies that **strcmp** yields the same result as **strcoll** when it is used to compare two strings transformed by **strxfrm**.

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

main()
{
   char *str1, *str2,                      /* input strings pointers */
   txf1[80],  txf2[80];                    /* transform arrays       */
   int result_strcoll, result_strcmp;      /* compare results        */

   str1 = " A B C D";
   str2 = " A C B";

   if ((strxfrm(txf1, str1, sizeof(txf1)) < sizeof(txf1)) &&
       (strxfrm(txf2, str2, sizeof(txf2)) < sizeof(txf2)))
      result_strcmp = strcmp(txf1, txf2);
   else exit(4);                  /* error exit if length is too big */

   result_strcoll = strcoll(str1, str2);    /* Get strcoll result.    */

      /* Result must be 0 or result signs must be the same.           */
   if ((result_strcmp == result_strcoll) ||
       (result_strcmp*result_strcoll > 0))
      exit(0);
   else exit(8);                           /* Else this is an error. */
}
```

User-Added Locales

This section discusses how to supplement the standard locales (**''S370''** and **''POSIX''**) and the three nonstandard locales supplied by the SAS/C Library by creating your own locales. Following is a discussion of the user-added structures that correspond to the standard locale structures, a listing of the header file **<localeu.h>** and an example locale (**''SAMP''**), and discussions of user-defined **strcoll** and **strxfrm** functions.

Creating a New Locale

Creating a new locale involves two steps:

- collecting locale-specific information
- compiling and linking locale-specific routines.

Once these tasks are completed, the necessary locale information in object form can be properly loaded, enabling a program to use the locale with a call to **setlocale**.

The header file **<localeu.h>** maps the various data structures and routines required for a locale. You must include it as well as **<locale.h>** when compiling a locale. Locale source code should be compiled with the **RENT** or **RENTEXT** compiler option and link-edited **RENT** (for re-entrant).

Each category for **setlocale** has a structure mapped within **<localeu.h>** that corresponds to it. The following table describes the user-supplied categories that correspond to the categories for **setlocale**.

Table 10.1 *User-Supplied Locale Categories*

Category	Structure Name	Description
`LC_NUMERIC`	`_lc_numeric`	`LC_NUMERIC` contains nonmonetary numeric formatting items; see the description of `localeconv` in Chapter 15, "Localization Functions."
`LC_MONETARY`	`_lc_monetary`	`LC_MONETARY` contains monetary formatting items; see the description of `localeconv` in `localeconv` in Chapter 15.
`LC_TIME`	`_lc_time`	`LC_TIME` contains function pointers to locale-specific date and formatting routines, pointers to month and weekday names, and a.m. and p.m. designation; all are used with various `strftime` formats.
`LC_CTYPE`	`_lc_ctype`	`LC_CTYPE` contains a flag for enablement of DBCS processing and string recognition by other library routines, including recognition of multibyte characters in formatted I/O format strings such as those used in `printf`. `LC_CTYPE` also contains a pointer to the character type table that affects the behavior of the character type functions such as `isalpha` and `tolower`.
`LC_COLLATE`	`_lc_collate`	`LC_COLLATE` contains a mode flag indicating the processing mode and collation table pointer for `strxfrm` and `strcoll`. Mode flag meanings are as follows: `0` indicates single-byte mode. `1` indicates double-byte mode. `>1` indicates multibyte mode. For single-byte mode, the library's `strxfrm` and `strcoll` functions use the collation table if supplied. In double-byte and multibyte mode, any use of the table is strictly left up to the user-supplied routines. The library functions use a standard double-byte collation in double-byte mode when the collation table pointer is `NULL`. Included in this category are function pointers to locale-specific versions of `strxfrm` and `strcoll`. The locale-specific version of the `strxfrm` function requires an additional fourth parameter. This parameter is a pointer to a `size_t` variable where the number of characters consumed from the input string by the function is placed. Also, the value returned by the locale's `strxfrm` is the number of characters placed in the output array, not necessarily the total transformed string length.

`LC_ALL`	`void *_lc_all[5]`	`LC_ALL` is an array of pointers to the other `_lc` structures in the following order:
		
		`[0] &_lc_collate`
		`[1] &_lc_ctype`
		`[2] &_lc_monetary`
		`[3] &_lc_numeric`
		`[4] &_lc_time`.

When the pointer to a structure that corresponds to a category is **NULL**, the name returned by **setlocale** reflects the new locale's name. However, it has the default "**C**" locale characteristics for that category. Similarly, if individual elements of a structure (pointers) are **NULL** or binary **0**, that piece of the locale also exhibits "**C**" locale behavior.

The <localeu.h> Header File

The following is a listing of the **<localeu.h>** header file required for compiling a user-added locale.

```
/* This header file defines additions to the ANSI locale.h header  */
/* file that are required for compiling both user-added  locale    */
/* value table load modules and several library functions. The     */
/* "C" defaults appear as comments for the _lc_numeric and         */
/*  _lc_monetary categories.                                       */
static struct _lc_numeric {
   char *decimal_point;         /* "."      */
   char *thousands_sep;         /* ""       */
   char *grouping;              /* ""       */
};

static struct _lc_monetary  {
   char *int_curr_symbol;       /* ""       */
   char *currency_symbol;       /* ""       */
   char *mon_decimal_point;     /* ""       */
   char *mon_thousands_sep;     /* ""       */
   char *mon_grouping;          /* ""       */
   char *positive_sign;         /* ""       */
   char *negative_sign;         /* ""       */
   char int_frac_digits;        /* CHAR_MAX */
   char frac_digits;            /* CHAR_MAX */
   char p_cs_precedes;          /* CHAR_MAX */
   char p_sep_by_space;         /* CHAR_MAX */
   char n_cs_precedes;          /* CHAR_MAX */
   char n_sep_by_space;         /* CHAR_MAX */
   char p_sign_posn;            /* CHAR_MAX */
   char n_sign_posn;            /* CHAR_MAX */
};

static const struct _lc_time  {
/* locale's date and time conversion routine                       */
   char *(*_lct_datetime_conv)();
/* address of locale's day conversion routine                      */
   char *(*_lct_date_conv)();
/* address of locale's time conversion routine                     */
   char *(*_lct_time_conv)();
/* address of weekday abbreviation table                           */
   char *_lct_wday_name [7] ;
/* address of full weekday name table                              */
   char *_lct_weekday_name [7] ;
/* address of month abbreviation table                             */
   char *_lct_mon_name [12] ;
/* address of full month name table                                */
   char *_lct_month_name [12] ;
```

```
/* locale's before-noon designation                                  */
   char *_lct_am;
/* locale's after-noon designation                                   */
   char *_lct_pm;
};

#define SBCS 0          /* single-byte character set                 */
#define DBCS 1          /* double-byte character set                 */

static const struct _lc_collate {
/* single-, double-, or multibyte character indicator                */
   int _lcc_cmode;
/* pointer to collation table                                        */
   void *_lcc_colltab;
/* pointer to user-added strcoll function                            */
   int (*_lcc_strcoll)();
/* pointer to user-added strxfrm function                            */
   size_t (*_lcc_strxfrm)();
};

static const struct _lc_ctype {
/* single-, double-, or multibyte character indicator                */
   int _lcc_cmode;

/* character type table pointer                                      */
   void *_lcc_ctab;
};

/* If _lcc_cmode is set to DBCS, it only has an impact on the ANSI */
/* multibyte character handling functions, not on isalpha, and     */
/* so on. _lcc_ctab is for single-byte characters only, per the    */
/* ANSI ctype.h-allowed representation of "unsigned char," and     */
/* has no relation to to _lcc_mode.                                */


static const void *_lc_all [5] ;        /* pointers to _lc struct */
                                        /* [0]  -  &_lc_collate   */
                                        /* [1]  -  &_lc_ctype     */
                                        /* [2]  -  &_lc_monetary  */
                                        /* [3]  -  &_lc_numeric   */
                                        /* [4]  -  &_lc_time      */
```

Example Locales

Example locales **L$CLSAMP** ("**SAMP**") and **L$CLDBEX** ("**DBEX**") are provided in source form with the compiler and library to serve as *skeleton* locales. You can easily modify these locales to create new locales. Ask your SAS Software Representative for SAS/C compiler products for information about obtaining copies of these programs. Here is an abbreviated listing of the "**SAMP**" locale, illustrating the data structures and routine formats required for a locale. The **L$CLDBEX** example is a double-byte example locale (not shown) with sample **strcoll** and **strxfrm** routines.

The “SAMP” locale

```
#title l$clsamp -- "SAMP" sample locale

/* This is the "SAMP" locale value module table, which      */
/* provides a skeleton example to modify for a              */
/* particular locale.  For those locales requiring          */
/* double-byte character support, see the "DBEX" locale     */
/* (L$DLDBEX) for examples of setting up a double-byte      */
/* LC_CTYPE, LC_COLLATE, strcoll, and strxfrm.              */
/*                                                          */
/* Any addresses of functions or tables not specified       */
/* with a category use the "C" locale equivalent            */
/* function or table. If a whole category is not specified  */
/* and the locale is requested for that category with       */
/* setlocale, effectively the "C" locale is used, although  */
/* the locale string returned contains the locale's name.   */


#include <stddef.h>
#include <locale.h>
#include <localeu.h>
#include <dynam.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#eject

/*                                                          */
/* ENTRY:  <=== _dynamn (externally visible) needs to be    */
/*             be compiled with SNAME l$clsam.              */
/*                                                          */
/* USAGE:  <=== Prototype call: _dynamn (dynamically        */
/*             loaded with the loadm function from the      */
/*             setlocale function).                         */
/*                                                          */

/*             Needs to be compiled with RENT or RENText    */
/*             option and link-edited RENT.                 */
/*                                                          */
/*             For example, the following is a call made    */
/*             to setlocale:                                */
/*                setlocale(LC_ALL, "SAMP");                */
/*             The following code is executed within        */
/*             setlocale code to load L$CLSAMP and then     */
/*             call it:                                     */
/*                loadm(L$CLSAMP, &fp)                      */
/*                fncptr = (char *** )(*fp)();              */
/*                                                          */
/* ARGUMENTS:  <=== None                                    */
/*                                                          */
/* RETURNS:    <=== A pointer to an array of pointers       */
/*                                                          */
```

```
/* static const void *lc_all_samp[5]  =                    */
/*    &collate,             collate pointer                */
/*    &ctype,               ctype pointer                  */
/*    &monetary             monetary pointer               */
/*    &numeric              numeric pointer                */
/*    &time                 time format pointer            */
/*                                                         */
/*                                                         */
/* END                                                     */

#eject

/*-----------------COLLATION category----------------------*/

static const unsigned char sbcs_collate_table_samp [256]  =

/* If a collation table is specified, that is, its address  */
/* is nonzero, a locale strxfrm function is not coded, and  */
/* the locale is not a multibyte (double-byte) locale, then */
/* the collation array must have 256 elements that          */
/* translate any character's 8-bit representation to its    */
/* proper place in the locale's collating sequence.         */

0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,  /* 0x00-0f */
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,

0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,  /* 0x10-1f */
0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,

0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,  /* 0x20-2f */
0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,

.     .     .     .     .     .     .     .      /* 0x30-ef */

.     .     .     .     .     .     .     .

.     .     .     .     .     .     .     .

0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7, /* 0xf0-ff */
0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff;

#define SBCS 0
#define DBCS 1

static const struct _lc_collate lc_collate_samp  =  {
   SBCS,                    /* single-byte character mode */
   sbcs_collate_table_samp, /* collation table address    */
   0,                       /* locale strcoll collation   */
                            /* function pointer           */
   0                        /* locale strxfrm transform   */
                            /* function pointer           */
   ;
```

```
/* See L$CLDBEX for DBCS example of strcoll and strxfrm functions. */

#eject

/*-----------------CTYPE category--------------------------*/

#define U 1    /* uppercase       */
#define L 2    /* lowercase       */
#define N 4    /* number          */
#define W 8    /* white space     */
#define P 16   /* punctuation     */
#define S 32   /* blank           */
#define AX 64  /* alpha extender  */
#define X 128  /* hexadecimal     */

static const unsigned char lc_ctab_samp[513]  =
{

/* The character type table array, if coded, must contain  */
/* 513 single char elements. The first element is the EOF  */
/* representation (-1 or 0xff) followed by 256 elements    */
/* that contain the char types for any 8-bit character     */
/* returned by functions isalpha, isnumeric, and so on.    */
/* The next 256 elements contain the mappings for the      */
/* tolower and toupper string transformation functions.    */

0,       /* -1 = EOF */
0,       /* 00 = nul */
0,       /* 01 = soh */
0,       /* 02 = stx */
0,       /* 03 = etx */
0,       /* 04 = sel */
W,       /* 05 = ht  */
0,       /* 06 = rnl */
0,       /* 07 = del */
0,       /* 08 = ge  */
0,       /* 09 = sps */
0,       /* 0a = rpt */
W,       /* 0B = vt  */
W,       /* 0C = ff  */
W,       /* 0D = cr  */
0,       /* 0E = so  */
0,       /* 0F = si  */
0,       /* 10 = dle */
0,       /* 11 = dcl */
0,       /* 12 = dc2 */
0,       /* 13 = dc3 */
.        .
.        .
.        .
U,       /* E6 = W   */
U,       /* E7 = X   */
U,       /* E8 = Y   */
U,       /* E9 = Z   */
```

```
0,       /* EA       */
0,       /* EB       */
0,       /* EC       */
0,       /* ED       */
0,       /* EE       */
0,       /* EF       */
N|X,     /* F0 = 0   */
N|X,     /* F1 = 1   */
N|X,     /* F2 = 2   */
N|X,     /* F3 = 3   */
N|X,     /* F4 = 4   */
N|X,     /* F5 = 5   */
N|X,     /* F6 = 6   */
N|X,     /* F7 = 7   */
N|X,     /* F8 = 8   */
N|X,     /* F9 = 9   */
0,       /* FA       */
0,       /* FB       */
0,       /* FC       */
0,       /* FD       */
0,       /* FE       */
0,       /* FF = eo  */

   /* Lower 257 bytes contain char types, next */
   /* 256 contain the tolower and toupper      */
   /* character mappings.                      */
0x00,    /* 00 = nul */
0x01,    /* 01 = soh */
0x02,    /* 02 = stx */
0x03,    /* 03 = etx */
.        .
.        .
.        .
0x7d,    /* 7D = '           */
0x7e,    /* 7E = =           */
0x7f,    /* 7F = "           */
0x80,    /* 80               */
0xc1,    /* 81 = a -> C1 = A */
0xc2,    /* 82 = b -> C2 = B */
0xc3,    /* 83 = c -> C3 = C */
0xc4,    /* 84 = d -> C4 = D */
0xc5,    /* 85 = e -> C5 = E */
0xc6,    /* 86 = f -> C6 = F */
0xc7,    /* 87 = g -> C7 = G */
0xc8,    /* 88 = h -> C8 = H */
0xc9,    /* 89 = i -> C9 = I */
.        .
.        .
.        .
0xbc,    /* BC                       */
0xbd,    /* BD = ]  (close bracket) */
0xbe,    /* BE                       */
0xbf,    /* BF                       */
0xc0,    /* C0 =  (open brace)       */
```

```
0x81,    /* C1 = A -> 81 = a        */
0x82,    /* C2 = B -> 82 = b        */
0x83,    /* C3 = C -> 83 = c        */
0x84,    /* C4 = D -> 84 = d        */
0x85,    /* C5 = E -> 85 = e        */
0x86,    /* C6 = F -> 86 = f        */
0x87,    /* C7 = G -> 87 = g        */
0x88,    /* C8 = H -> 88 = h        */
0x89,    /* C9 = I -> 89 = i        */
0xca,    /* CA = shy                */
0xcb,    /* CB                      */
0xcc,    /* CC                      */
0xcd,    /* CD                      */
0xce,    /* CE                      */
.        .
.        .
.        .
0xf7,    /* F7 = 7   */
0xf8,    /* F8 = 8   */
0xf9,    /* F9 = 9   */
0xfa,    /* FA       */
0xfb,    /* FB       */
0xfc,    /* FC       */
0xfd,    /* FD       */
0xfe,    /* FE       */
0xff     /* FF = eo  */
};


static const struct _lc_ctype lc_ctype_samp =
   SBCS,             /* single-byte character mode */
   &lc_ctab_samp     /* ctype table pointer */
   };

#eject

/*------------NUMERIC category--------------------*/

const static struct _lc_numeric lc_numeric_samp = {
   ".",                     /* decimal_point */
   ",",                     /* thousands_sep */
   "\3"                    /* grouping */
};

/*------------MONETARY category---------------------*/

static const struct _lc_monetary lc_monetary_samp  = {
   "DOL",                   /* int_curr_symbol    */
   "$",                     /* currency_symbol    */
   ".",                     /* mon_decimal_point  */
   ",",                     /* mon_thousands_sep  */
   "\3",                    /* mon_grouping       */
   "",                      /* positive_sign      */
   "-",                     /* negative_sign      */
```

```
   2,                      /* int_frac_digits    */
   2,                      /* frac_digits        */
   1,                      /* p_cs_precedes      */
   0,                      /* p_sep_by_space     */
   1,                      /* n_cs_precedes      */
   0,                      /* n_sep_by_space     */
   1,                      /* p_sign_posn        */
   1                       /* n_sign_posn        */
   ;

#eject

/*------------TIME category-----------------------------*/

char *sampdcnv(struct tm *tp);

static const struct _lc_time  lc_time_samp =  {
   0,             /* pointer to date and time conversion */
                  /* routine function pointer            */
   &sampdcnv,     /* pointer to date conversion          */
                  /* routine function pointer            */
   0,             /* pointer to time conversion          */
                  /* routine function pointer            */
      /* weekday name abbreviations                       */
   "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat",
      /* weekday full names                               */
   "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday",
      /* month name abbreviations                         */
   "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
      /* month full names */
   "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November",
    "December",
   "AM",                  /* locale's "AM" equivalent  */
   "PM"                   /* locale's "PM" equivalent  */
};

char *sampdcnv(struct tm *tp)
{

/* SAMP date conversion routine          */
/* Function returns the date in the form: */
/* wkd mon dd 'yy                        */
/* for example, Thu Oct 10 '85.          */

char *time_format;

time_format = asctime(tp);

memcpy(time_format + 11, " '", 2);
memcpy(time_format + 13, time_format + 22, 2);
```

```
*(time_format + 15) = '\ 0';

return time_format;
 /* End sampdcnv.  */

#eject

/* ALL category - array of pointers to category structures */

static const void *lc_all_samp[5]  =
   &lc_collate_samp ,  /* pointer to collate category      */
   &lc_ctype_samp ,    /* pointer to ctype category        */
   &lc_monetary_samp , /* pointer to monetary category     */
   &lc_numeric_samp ,  /* pointer to numeric samp          */
   &lc_time_samp       /* pointer to time samp             */
   ;

/*-----------------Return category pointers---------------*/

void *_dynamn()  /* executable entry point */
{
   return (void *)&lc_all_samp;  /* Return address of ALL array. */
}
```

LOCALE strcoll EQUIVALENT

If the library **strcoll** function is not adequate for the needs of a locale, you can write and use your own routine to do the collation. You include this routine as part of the **LC_COLLATE** category of a locale to be called from the library **strcoll** function after **setlocale** has loaded the locale. Because it is your own routine, you can give it any legal name, as long as you are consistent in its use. (For instance, the following example uses the name **loclcoll**.)

The locale's routine can make use of any information available in the locale, such as the mode and collation tables. In addition, if the **LC_COLLATE** mode is not **0**, the collation tables coded as part of the locale are not restricted to any format as long as the locale's **strxfrm** and **strcoll** routines can understand them.

The locale's routine is invoked from the library's **strcoll** function with the equivalent of the following call:

```
      /* library strcoll function                  */
   int strcoll(char *str1, const char *str2)
   {
   .
   .
   .

      /* Return locale's strcoll value to library's */
      /* strcoll caller.                           */
   return loclcoll(str1, str2);
   }
```

The **loclcoll** function code appears as part of the **LC_COLLATE** category within the locale source code:

```
.
.
.
   /* collation tables, transformation tables,            */
   /* and other locale data                               */
int loclcoll(const char *str1, const char *str2)

/*   ARG    DCL              DESCRIPTION                  */
/*                                                        */
/*   str1   const char *    pointer to first input string */
/*                                                        */
/*   str2   const char *    pointer to second input string */
/*                                                        */
/*  RETURNS:  <=== str1 < str2      a negative value      */
/*                 str1 = str2      0                     */
/*                 str1 > str2      a positive value      */
/*                                                        */

{
.
. /* locale's equivalent strcoll function code          */
.
return x     /* Return a result, x.             */
}
   .
   . /* more locale data, routines, and so on           */
   .
```

For an example of a locale routine for **strcoll**, see the **L$CLDBEX** source code member distributed with the compiler and library.

LOCALE strxfrm EQUIVALENT

If the library **strxfrm** function is not adequate for the needs of a locale, you can write and use your own routine to do the transformation. You include this routine as part of the **LC_COLLATE** category of a locale to be called from the library **strxfrm** function after being loaded with an appropriate **setlocale** call. Because it is your own routine, you can give it any legal name, as long as you are consistent in its use. (For instance, the following example uses the name **loclxfrm**.)

There is one main difference in behavior requirements for the locale equivalent of the **strxfrm** function and behavior requirements for the library version. After the output buffer is filled, it stops scanning the input string and returns the size of the filled output buffer rather than the total transformed length. It also places the number of characters consumed from the input string in the area addressed by an additional fourth parameter. The locale's routine can use any information available in the locale, such as the mode and collation tables. In addition, if the **LC_COLLATE** mode is not **0**, the collation and transformation tables coded as part of the locale are not restricted to any format as long as the locale's **strxfrm** and **strcoll** routines can understand them.

The reason the behavior requirements of the library and locale **strxfrm** routines differ is to allow the **strcoll** function to call **strxfrm** with a limited buffer that might permit only partial transformation of the whole string. Theoretically, any

number of output characters can be produced by **strxfrm** from any number of input characters.

The locale's **strxfrm** routine is invoked from within **strxfrm** with an equivalent of the following call. (The choice of **``loclxfrm''** is arbitrary. It could be any legal name as long as it is consistent.)

```
size_t strxfrm(char *str1, const char *str2, size_t n)
{
      size_t nchar_xfrmed, used;
   .
   .
   .
   wchar_xfrmed = loclxfrm(str1, str2, n, &used);
   .
   .
   .
}
```

The **loclxfrm** function code appears as part of the **LC_COLLATE** category within the locale source code:

```
.
. /* collation tables, transformation tables, and other   */
  /* locale data                                          */
.
size_t loclxfrm(char *str1, const char *str2,
                size_t n, size_t *used)

/* ARG    DCL            DESCRIPTION                     */
/*                                                       */
/* str1   char *         pointer to the transformed      */
/*                       string output array             */
/*                                                       */
/* str2   const char *   pointer to input string array   */
/*                                                       */
/* n      size_t         maximum number of bytes         */
/*                       (characters) written to str1    */
/*                       including the terminating null. */
/*                       If n or more characters are     */
/*                       required for the transformed    */
/*                       string, only the first n are    */
/*                       written and the string is not   */
/*                       null-terminated.                */
/*                                                       */
/* used   size_t *       pointer to size_t (unsigned int) */
/*                       value where the number of input */
/*                       characters consumed from the    */
/*                       input string str2 is returned   */
/*                                                       */
/* RETURNS: <=== The number of characters placed in the  */
/*              output transformation array is returned. */
/*                                                       */
/*              Also, the number of characters consumed  */
/*              (scanned) from the input string str2 is  */
```

```
/*              placed in the size_t (unsigned int)      */
/*              value pointed to by used.                */
/*                                                       */
/*              The total number of characters required  */
/*              for the transformation is obtained by a  */
/*              special call:                            */
/*   total_loclxfrm_len = loclxfrm(0, s2, 0, &used)      */

{
.
. /* locale's strxfrm code                              */
}
.
.
. /* more locale data, routines, and so on              */
.
```

For an example of a locale routine for **strxfrm**, see the **L$CLDBEX** source code member distributed with the compiler and library.

11 Multibyte Character Functions

11-1 Introduction
11-1 SAS/C Implementation of Multibyte Character Sequences
11-1 Mixed DBCS Sequences
11-3 Pure DBCS Sequences
11-3 Converting Sequences
11-3 DBCS Support with SPE
11-3 Formatted I/O Functions and Multibyte Character Sequences
11-4 Locales and Multibyte Character Sequences
11-4 Function Descriptions

Introduction

This chapter introduces fundamental concepts of multibyte character sequences, discusses the SAS/C implementation of multibyte character sequences, and describes five functions designed specifically to work with multibyte character sequences. These functions are as follows:

`mblen`	determines the length of a multibyte character.
`mbstowcs`	converts a multibyte character sequence to a wide character sequence.
`mbtowc`	converts a single multibyte character to a wide character.
`wcstombs`	converts a wide character sequence to a multibyte character sequence.
`wctomb`	converts a single wide character to a multibyte character.

SAS/C Implementation of Multibyte Character Sequences

The ISO/ANSI C Standard defines a multibyte character as consisting of 1 or more bytes, but it leaves the implementation of multibyte sequences up to individual vendors. The SAS/C Library supports both single-byte and multibyte characters. Characters consisting of more than 1 byte are supported in the context of the EBCDIC Double-Byte Character Set (DBCS). "Multibyte Character Support" in Chapter 4, "Compiler Processing and Code Generation Conventions," of *SAS/C Compiler and Library User's Guide* discusses the SAS/C implementation of multibyte characters in more detail.

There are two kinds of DBCS sequences, *mixed* and *pure* (in Standard terminology, *multibyte* and *wide*; this discussion uses DBCS terms). Mixed sequences may contain both single- and double-byte characters, while pure sequences contain only double-byte characters.

Mixed DBCS Sequences

Several methods exist for handling mixed DBCS sequences. For example, an encoding scheme may set aside a subrange of values to signal multibyte sequences. Another popular encoding scheme sets aside a single byte value to indicate a *shift out* from a normal interpretation of character codes to an alternate interpretation, where groups of bytes represent certain characters. This method is referred to as *shift-out/shift-in* encoding and is the method the SAS/C Compiler uses to handle multibyte sequences. This encoding scheme uses *shift states*, which indicate how a byte value or set of byte

values will be interpreted. The SAS/C Compiler uses shift-out/shift-in encoding because it is the DBCS encoding defined for the EBCDIC character set.

A mixed DBCS sequence must follow these rules:

- DBCS sequences must begin and end in the *initial shift state*, that is, 1 byte per character.
- any subsequence of double-byte characters must be preceded with and followed by a *state-dependent encoding* , SO/SI (shift-out/shift-in).

  SO indicates a *shift out* from the normal single-byte interpretation to an alternative interpretation of characters.

  SI indicates a *shift in*, that is, a return to the usual single-byte interpretation.

  The hexadecimal value for SO is **\x0E** and the value for SI is **\x0F**. For example, the following is a mixed DBCS string in hex:

```
\x81\x82\x83\x0E\x41\x52\x0F\x81
```

  The **\x41\x52** between the **\x0E** and **\x0F** is a double-byte character. The other characters are single-byte.
- SO/SIs must be paired.
- SO/SIs cannot be nested.
- an SO/SI pair must surround an even number of bytes.
- the Standard requires that a null character terminate a multibyte sequence, even in the double-byte shift state. This is a departure from DBCS sequences in other languages, which always require an explicit shift back into the single-byte shift state before the end of a sequence.

In the single-byte state, each character is represented by 1 byte and has its EBCDIC value. For example, the character constant **'a'** is **\x81** in hex.

In the double-byte state, each character is represented by 2 bytes. Double-byte characters must conform to the following constraints:

- all first bytes must have values between **\x41** and **\xFE**, except for the encoding of the blank space.
- all second bytes must have values between **\x41** and **\xFE**, except for the encoding of the blank space.
- the blank space is represented by **\x40\x40**.

The SAS/C implementation of multibyte characters does not allow empty SO/SI pairs (**\x0E\x0F**). For example, the following sequence (in hex), which might be construed as a single multibyte character, is not valid:

```
\x0E\x0F\x0E\x0F\x0E\x0F\x0E\x41\x81\x0F
```

This restriction is imposed because the number of bytes used to represent a multibyte character would, in theory, be unbounded; but the Standard requires an implementation to define a maximum byte-length for a multibyte character.

On the other hand, consecutive SI/SO pairs (**\x0F\x0E**) are permitted because they may result from string concatenation. For example, the following sequence (in hex) is valid:

```
\x0E\x41\x81\x0F\x0E\x41\x83\x0F
```

Pure DBCS Sequences

Pure DBCS sequences contain only double-byte characters. Thus, no SO/SI pairs are needed. The Standard supports pure sequences by providing a type capable of holding wide characters. This type, **wchar_t**, is implementation-defined as an integer type capable of representing all the codes for the largest character set in locales supported by the implementation. **wchar_t** is implemented by the SAS/C Library in **<stddef.h>** as follows:

```
typedef unsigned short wchar_t;
```

Converting Sequences

When converting from mixed to pure, all SO/SI pairs are removed from the sequence, and the double-byte characters are moved into corresponding **wchar_t** elements. When a mixed character sequence contains characters that require only a single byte, these characters are converted to **wchar_t**, but their values are unchanged. For example, the mixed string (**''abc''**) is represented as follows:

```
\x81\x82\x83\x00
```

When converted to a pure DBCS sequence, the string will become the following:

```
\x00\x81\x00\x82\x00\x83\x00\x00
```

Use the **mbtowc** function to convert 1 multibyte character to a double-byte character. Use the **mbstowcs** function to convert a sequence of multibyte characters to a double-byte sequence. Note that this function assumes the sequence is terminated by the null character, **\x00**. You also can use regular string-handling functions with mixed DBCS sequences. For example, you can use **strlen** to determine the byte-length of a sequence, as long as the sequence is null-terminated.

When converting from pure to mixed, SO/SI pairs are added to the sequence as necessary. Use the **wctomb** function to convert 1 double-byte character to a multibyte character. Use the **wcstombs** function to convert a sequence of double-byte characters to a multibyte sequence. Note that this function assumes the sequence is terminated by the null wide character, **\x00\x00**.

DBCS Support with SPE

The multibyte character functions can be used with the SPE framework. Normally this framework does not support locales, and by default DBCS support is not enabled. To enable DBCS support with SPE, turn on the **CRABDBCS** bit in **CRABFLGM** in your start-up routine or in L$UMAIN.

Formatted I/O Functions and Multibyte Character Sequences

Mixed DBCS sequences are supported in the format string for the formatted I/O functions such as **printf**, **sprintf**, **scanf**, **sscanf**, and **strftime** as required by the Standard. Recognition of a mixed sequence within a format requires that a double-byte locale such as **''DBCS''** be in effect. Mixed sequences are treated like any other character sequence in the format string with one exception; they are copied unchanged to output or matched on **scanf** input, but invalid sequences may cause premature termination of the function. The conversion specifier **%** and specifications associated with it, which are imbedded within the format string, are recognized only while in single-byte mode, which is the initial shift state at the beginning of the format string.

Locales and Multibyte Character Sequences

The processing of multibyte character sequences is dependent on the current locale. (See Chapter 10, "Localization" on page 10-1 for a full discussion of locales.) For example, some locales support DBCS sequences and some do not. The standard locales **''S370''** and **''POSIX''** do not support DBCS sequences. The default locale, **''''**, may or may not support DBCS sequences, depending on the values of locale-related environment variables. Of the three locales supplied by the SAS/C Library, **''DBCS''** and **''DBEX''** support DBCS sequences, while **''SAMP''** does not.

The macro **MB_CUR_MAX**, defined in **<stdlib.h>**, defines the longest sequence of bytes needed to represent a single multibyte character in the current locale. The macro **MB_LEN_MAX**, on the other hand, is not locale-dependent and defines the longest multibyte character permitted across *all* locales.

Function Descriptions

Descriptions of each multibyte character function follow. Each description includes a synopsis, a description, discussions of return values and portability issues, and an example. Also, errors, cautions, diagnostics, implementation details, and usage notes are included if appropriate. None of the multibyte character functions are supported by traditional UNIX C Compilers.

mblen Determine Length of a Multibyte Character

SYNOPSIS

```
#include <stdlib.h>

int mblen(const char *s, size_t n);
```

DESCRIPTION

mblen determines how many bytes are needed to represent the multibyte character pointed to by **s**.

n specifies the maximum number of bytes of the multibyte character sequence to examine.

RETURN VALUE

If **s** is not **NULL**, the return value is as follows:

0	is returned if **s** points to the null character.
length of the multibyte character	is returned if the next **n** or fewer bytes constitute a valid multibyte character.
−1	is returned if the next **n** or fewer bytes do not constitute a valid multibyte character.

If **s** is **NULL** , the return value is as follows:

nonzero value	is returned if the current locale supports state-dependent encodings.
0	is returned if the current locale does not support state-dependent encodings.

CAUTIONS

A diagnostic is not issued if **mblen** encounters invalid data; a return value of **−1** is the only indication of an error.

EXAMPLE

```
   /* This example counts multibyte characters (not including   */
   /* terminating null) in a DBCS mixed string using mblen().   */

#include <locale.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
```

mblen Determine Length of a Multibyte Character
(continued)

```
   /* "strptr" points to the beginning of a DBCS MIXED string.  */
   /* RETURNS: number of multibyte characters                   */
int count1(char *strptr)
{
   int i = 0;          /* number of multibyte characters found  */
   int charlen;        /* byte length of current characte       */

   /* Inform library that we will be accepting a DBCS string.   */
   /* That is, SO and SI are not regular control characters:    */
   /* they indicate a change in shift state.                    */
   setlocale(LC_ALL, "dbcs");
      /* Reset to initial shift state. (A valid mixed string    */
      /* must begin in initial shift state).                    */
   mblen(NULL, 0);

     /* One loop iteration per character. Advance "strptr" by   */
     /* number of bytes consumed.                               */
   while (charlen = mblen(strptr, MB_LEN_MAX)) {
      if (charlen < 0) {
         fputs("Invalid MIXED DBCS string", stderr);
         abort();
         fclose(stderr);
      }
      strptr += charlen;
      i++;
   }
   return i;
}
```

mbstowcs Convert a Multibyte Character Sequence to a Wide Sequence

SYNOPSIS

```
#include <stdlib.h>

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

DESCRIPTION

mbstowcs converts a sequence of multibyte characters (mixed DBCS sequence) pointed to by **s** into a sequence of corresponding wide characters (pure DBCS sequence) and stores the output sequence in the array pointed to by **pwcs**. The multibyte character sequence is assumed to begin in the initial shift state.

n specifies the maximum number of wide characters to be stored.

RETURN VALUE

If the multibyte character sequence is valid, **mbstowcs** returns the number of elements of **pwcs** that were modified, excluding the terminating **0** code, if any. If the sequence of multibyte characters is invalid, **mbstowcs** returns **−1**.

CAUTIONS

No multibyte characters that follow a null character are examined or converted. If the sequence you want to convert contains such a value in the middle, you should use a loop that calls **mbtowc**.

If copying takes place between objects that overlap, the behavior of **mbstowcs** is undefined.

A diagnostic is not issued if **mbstowcs** encounters invalid data; a return value of **−1** is the only indication of an error.

EXAMPLE

This example replaces all occurences of a given character in a mixed DBCS string. The string is assumed to have a maximum length of 81 characters. This example uses **mbstowcs** and **wcstomb**.

```
#include <locale.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_CHARACTERS 81
   /* "old_string" is the input MIXED DBCS string. "new_string" */
   /* is the output MIXED DBCS string. "old_wchar" is the       */
   /* multibyte character to be  replaced. "new_wchar" is the   */
   /* multibyte character to replace with.                      */
void mbsrepl(char *old_string, char *new_string,
             wchar_t old_wchar, wchar_t new_wchar)
```

mbstowcs Convert a Multibyte Character Sequence to a Wide Sequence
(continued)

```
{
   wchar_t work[MAX_CHARACTERS];
   int nchars;
   int i;

      /* Inform library that we will be accepting a DBCS string.*/
      /* That is, SO and SI are not regular control characters: */
      /* they indicate a change in shift state.                 */
   setlocale(LC_ALL, "dbcs");

   nchars = mbstowcs(work, old_string, MAX_CHARACTERS);
   if (nchars < 0) {
      fputs("Invalid DBCS string.\n", stderr);
      fclose(stderr);
      abort();
   }

      /* Perform the actual substitution.                       */
      for (i = 0; i < nchars; i++)
          if (work[i] ==  old_wchar)
              work[i] = new_wchar;

      /* Convert back to MIXED format.                          */
   nchars = wcstombs(new_string, work, MAX_CHARACTERS);

      /* See if the replacement caused the string to overflow.  */
   if (nchars == MAX_CHARACTERS) {
      fputs("Replacement string too large.\n", stderr);
      abort();
      fclose(stderr);
   }
}
```

mbtowc Convert a Multibyte Character to a Wide Character

SYNOPSIS

```
#include <stdlib.h>

int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

DESCRIPTION

mbtowc determines how many bytes are needed to represent the multibyte character pointed to by **s**. If **s** is not **NULL**, **mbtowc** then stores the corresponding wide character in the array pointed to by **pwc**.

n specifies the maximum number of bytes to examine in the array pointed to by **pwc**.

RETURN VALUE

If **s** is not **NULL**, the return value is as follows:

0	is returned if **s** points to the null character.
length of the multibyte character	is returned if the next **n** or fewer bytes constitute a valid multibyte character.
−1	is returned if the next **n** or fewer bytes do not constitute a valid multibyte character.

If **s** is **NULL**, the return value is as follows:

nonzero value	is returned if the current locale supports state-dependent encodings.
0	is returned if the current locale does not support state-dependent encodings.

CAUTIONS

A diagnostic is not issued if **mbtowc** encounters invalid data; a return value of **−1** is the only indication of an error.

EXAMPLE

This example finds a multibyte character in a mixed DBCS string using **mbtowc**.

```
#include <locale.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>
```

mbtowc Convert a Multibyte Character to a Wide Character
(continued)

```
   /* "begstr" points to the beginning of a DBCS MIXED string.  */
   /*  "mbc_sought" is the character value we're looking for.   */
int mbfind(char *begstr, wchar_t int mbc_sought)
{
   int mbclen;        /* length (in bytes) of current character */
   wchar_t mbc;       /* value of current character             */
   char *strptr;      /* pointer to current location in string  */
   strptr = begstr;

      /* Inform library that we will be accepting a DBCS string.*/
      /* That is, SO and SI are not regular control characters: */
      /* they indicate a change in shift state.                 */
   setlocale(LC_ALL, "dbcs");

      /* Reset to initial shift state. (A valid mixed string    */
      /* must begin in initial shift state).                    */
   mbtowc((wchar_t *)NULL, NULL, 0);

      /* One loop iteration per character. Advance "strptr" by  */
      /* number of bytes consumed.                              */
   while (mbclen = mbtowc(&mbc, strptr, MB_LEN_MAX)) {
      if (mbclen < 0) {
         fputs("Invalid pure DBCS string\n", stderr);
         abort();
      }
      if (mbc == mbc_sought)
         break;
      strptr += mbclen;
   }

      /* Last character was not '\0' -- must have found it     */
   if (mbclen) {
      printf("MBFIND: found at byte offset %d\n", strptr - begstr);
      return 1;
   }
   else {
      puts("MBFIND: character not found\n");
      return 0;
   }
}
```

wcstombs Convert a Wide Character Sequence to a Multibyte Sequence

SYNOPSIS

```
#include <stdlib.h>

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

DESCRIPTION

wcstombs converts a sequence of wide characters (pure DBCS sequence) to a sequence of multibyte characters (mixed DBCS sequence). The wide characters are in the array pointed to by **pwcs**, and the resulting multibyte characters are stored in the array pointed to by **s**. The resulting multibyte character sequence begins in the initial shift state.

n specifies the maximum number of bytes to be filled with multibyte characters. The conversion stops if a multibyte character would exceed the limit of **n** bytes or if a null character is stored.

RETURN VALUE

If the multibyte character sequence is valid, **wcstombs** returns the number of bytes of **s** that were modified, excluding the terminating `0` byte, if any. If the sequence of multibyte characters is invalid, **wcstombs** returns **−1**.

CAUTIONS

If copying takes place between objects that overlap, the behavior of **wcstombs** is undefined.

A diagnostic is not issued if **wcstombs** encounters invalid data; a return value of `−1` is the only indication of an error.

EXAMPLE

See the example for **mbstowcs**.

wctomb Convert a Wide Character to a Multibyte Character

SYNOPSIS

```
#include <stdlib.h>

int wctomb(char *s, wchar_t wchar);
```

DESCRIPTION

wctomb determines how many bytes are needed to represent the multibyte character corresponding to the wide (pure DBCS) character whose value is **wchar**, including any change in shift state. It stores the multibyte character representation in the array pointed to by **s**, assuming **s** is not **NULL**. If the value of **wchar** is **0**, **wctomb** is left in the initial shift state.

RETURN VALUE

If **s** is not **NULL**, the return value is the number of bytes that make up the multibyte character corresponding to the value of **wchar**.

If **s** is **NULL**, the return value is as follows:

nonzero value	is returned if the current locale supports state-dependent encodings.
0	is returned if the current locale does not support state-dependent encodings.

The return value is never greater than the value of the **MB_CUR_MAX** macro.

CAUTIONS

A diagnostic is not issued if **wctomb** encounters invalid data; a return value of **-1** is the only indication of an error.

EXAMPLE

This example converts a PURE DBCS string to MIXED, stopping at the first new-line character. This example uses **wctomb**.

```
#include <stdlib.h>
#include <locale.h>
#include <limits.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_CHARACTERS 81

   /* "pure_string" is the input PURE DBCS string.              */
   /* "mixed_string" the output MIXED DBCS string.              */
void mbline(wchar_t *pure_string, char *mixed_string)
{
   int i;
   int mbclen;
   wchar_t wc;
```

wctomb Convert a Wide Character to a Multibyte Character
(continued)

```
   /* Inform library that we will be accepting a DBCS string. */
   /* That is, SO and SI are not regular control characters:  */
   /* they indicate a change in shift state.                  */
setlocale(LC_ALL, "dbcs");

wctomb(NULL, 0);            /* Reset to initial shift state. */

   /* One loop iteration per character. Advance "mixed_string"*/
   /* by number of bytes in character.                        */
i = 0;
do {
   wc = pure_string[i++];
   mbclen = wctomb(mixed_string, wc);
   if (mbclen < 0) {
      puts("Invalid PURE DBCS string.\n");
      abort();
      fclose(stdout);
   }
   mixed_string += mbclen;
} while (wc != L'\n');

*mixed_string = '\0';
}
```

12 User-Added Signals

12-1 Introduction
12-2 Notes on Handlers for User-Defined Signals
12-2 Restrictions on Synchronous and Asynchronous Signals
12-2 Summary of Routines for Adding Signals
12-4 Initialization Routine and sigdef Function
12-6 Signal Generator Routine
12-6 Default Routine
12-6 Interrupt Control Routine
12-7 Executive Routine
12-8 Final Routine
12-8 Jump Intercept Routine
12-8 SAS/C Library Routines for Adding New Signals
12-9 Routine for Synchronous Signals: L$CZSYN
12-10 Routine for Asynchronous Signals: L$CZENQ

Introduction

The SAS/C Library enables you to add new user signals in two ways. If you want to raise a user signal within your program to indicate an unusual or error condition, you can use the **`raise`** or **`siggen`** function. This type of user signal requires no special coding, only the use of these two functions. The default action for a user signal raised in this manner is always to ignore the signal. Refer to the descriptions of the **`raise`** and **`siggen`** functions for more information.

If, however, you want to raise a signal as a result of a software interrupt (such as an end-of-subtask notification) or a hardware interrupt (such as an I/O interrupt), you can do so by coding two or more routines to intercept these interrupts and make them known to the library. The first of the required routines, called the *initialization routine*, defines the signal to the library. The other required routine, called the *signal-generating routine*, is an operating system exit that is invoked to raise the signal for the library when the interrupt occurs. Both of these routines are normally written in assembler language. Most of the other routines that define how to handle the signal can be written in either C or assembler language. Table 12.1 on page 12-3 indicates the recommended language for each routine and describes how to write the routines that add user signals to the library.

Note: This book assumes that you are adding a signal that communicates to your C program when a software or hardware interrupt occurs. You may have other uses for this facility that are not explicitly discussed in this book.

Some familiarity with MVS or CMS interrupt handling facilities is assumed. Also, thorough familiarity with SAS/C signal functions and signal-handling techniques is assumed. Refer to Chapter 5, "Signal-Handling Functions," in *SAS/C Library Reference, Volume 1* for more information.

Notes on Handlers for User-Defined Signals

When you add new signals, the full array of signal-handling functions, including **siginfo**, **sigprocmask**, and **sigsuspend**, can be used in your program to control handling of the new signals. In some cases, you may want to define the new signal so that it does not permit the use of **longjmp** or **return** in a handler. Refer to "Executive Routine" on page 12-7 for more information.

Restrictions on Synchronous and Asynchronous Signals

User signals can be synchronous or asynchronous; you can define as many as eight of each type. Synchronous signals generally must be handled immediately, while asynchronous signals either do not need to be handled immediately or cannot be handled immediately for technical reasons.

Synchronous signals must be assigned one of the signal numbers defined by the symbols SIGUSR1 through SIGUSR8. Asynchronous signals must be assigned one of the signal numbers defined by the symbols SIGASY1 through SIGASY8. When you write the routine to define the signal, you can rename it to have a more useful and mnemonic name.

There are no special requirements for defining asynchronous signals. Additional considerations may apply, however, when defining asynchronous signals for programs that use **pause**, **sigpause**, **sigsuspend**, or **sleep**; refer to the description of the ECBPP field of ZENQARGS in "Routine for Asynchronous Signals: L$CZENQ" on page 12-10.

The following list describes the requirements for defining a signal as synchronous.

- The address of the CRAB must be in register 12, and the value in register 13 must be addressable at the time of the interrupt. If the optimized or minimal form of function linkage is in use, this register 13 value must address a C dynamic save area (DSA). Refer to Chapter 9, "Run-Time Argument Processing," in *SAS/C Compiler and Library User's Guide* for more information on the **=optimize** and **=minimal** options.
- The signal must not be able to occur while the SAS/C Library prolog or epilog is running.
- The SAS/C Library must be able to operate normally when the signal is generated. In particular, it must be possible to issue SVC instructions.
- Because of the possibility that the user may issue a call to **longjmp**, the program must be able to resume execution at some point other than the point of interrupt. (It is possible, though not recommended, to use the executive routine, described later in this chapter, to prevent using **longjmp** successfully by the handler.)

Summary of Routines for Adding Signals

To add a user signal to the library, you must write at least two routines. (Depending on your needs, more routines may be necessary.) Table 12.1 on page 12-3 lists the routines you may need to write to define and control a new signal. The table also indicates the preferred language for each routine. Most routines can be written in either C or assembler language.

Assembler routines called by the SAS/C Library rather than by the operating system can use the **CENTRY** and **CEXIT** macros for function linkage. (Refer to Chapter 11, "Communication with Assembler Programs," in *SAS/C Compiler and Library User's Guide*, for information on the **CENTRY** and **CEXIT** macros.)

Assembler routines that call C functions must use the `CENTRY` and `CEXIT` macros to run successfully when either the optimized or minimal form of function linkage is in use. Refer to Chapter 9 of *SAS/C Compiler and Library User's Guide* for more information on the `=optimize` and `=minimal` options.

Note: A routine that calls operating system macros is usually written in assembler language.

Table 12.1
Routines for Adding Signals

Routine	Required?/ Frequency of use?	Recommended Language*	Description
initialization routine	yes/ always	assembler	calls `sigdef` to define the new signal and identify which of the following routines are coded.
signal generator	yes/ always	assembler	intercepts the hardware or software interrupt and informs the library that the signal occurred. The routine is usually an operating system exit.
default handler	no/ common	C	establishes default actions that occur when no handler is defined for the new signal.
interrupt control	no/ seldom	assembler	communicates to the operating system when a signal is ignored, blocked, or handled in the default manner.
final routine	no/ very common	assembler	cancels signal handling at the end of the program. The routine is invoked after all files are closed.
executive routine	no/ seldom	C	supervises linkage to handlers. For example, the routine can prevent use of `longjmp` or normal returns.
jump intercept routine	no/ rare	assembler	notifies the operating system that the interrupt has been handled; that is, it clears the interrupt.

*Note that all of these routines except the jump intercept routine can be coded in either C or assembler.
This table simply indicates which language is preferred.

Initialization Routine and sigdef Function

To add a signal to the library, you must code an initialization routine. This routine is usually coded in assembler language. The initialization routine calls the **sigdef** function to define the signal. The initialization routine also calls an operating system macro to establish the address of the operating system exit that should be invoked when the signal occurs. For example, to define a CMS I/O interrupt signal called **SIGIOI**, the initialization routine may issue calls similar to the ones here. The call to **sigdef** in assembler might look like the following:

```
          LA   R1,DEFPARMS
          L    R15,=V(SIGDEF)
          BALR R14,R15       sigdef(SIGASY1,0,0,0,0,"IOI");

DEFPARMS DC   A(SIGASY1)    symbol definition obtained by
*                           "COPY SIGNALS"
         DC   4A(0)
         DC   A(SIGNAME)

SIGNAME  DC   C'IOI',X'00'
```

The call to **sigdef** renames SIGASY1 to SIGIOI but does not define any special routines for processing the signal (indicated by the 0s for the second DC in the DEFPARMS area).

A sample call to the CMS HNDINT macro to handle the interrupt might look like the following:

```
HNDINT SET,(TAP1,EXIT,unit,ASAP)
```

The call to HNDINT identifies the I/O unit number that causes the interrupt (**unit**) and the address of the operating system exit routine (**EXIT**) that you code to generate the signal for the library.

You can also use the initialization routine to save the C Run-Time Anchor Block (CRAB) address so that it can be accessed by the signal generator routine. (One way to do this is to request that the operating system provide the address as a user parameter to an exit routine.) Saving the address of the CRAB is frequently necessary because register 12 is dedicated to the CRAB address only during C program execution; it is usually not preserved by the operating system when an exit routine is called.

Note: It is possible to raise, handle, or block a user signal before it has been defined by a call to **sigdef**. When **sigdef** is called, it has no effect on the signal's status; that is, the signal remains blocked or unblocked, and any user handler remains in effect. However, if the call to **sigdef** defines a default handler, this default replaces the previous default handler.

sigdef Define User Signal

SYNOPSIS

```
#include <lcsignal.h>

int sigdef(int signum, void (*dfl)(int),
           int (*intcntl)(int, int, int, int),
           void (*executive)(int, char **, _HANDLER),
           void (*final)(int, char *name);
```

DESCRIPTION

The **signum** argument is the signal number. Specify the signal number as one of the codes, SIGUSR1 through SIGUSR8 or SIGASY1 through SIGASY8. Only one definition of each signal number is allowed.

Note: In an OpenEdition MVS application, **SIGUSR1** and **SIGUSR2** may be treated as OpenEdition signals rather than as SAS/C signals, depending on the use of the **oesigsetup** function and the way the program is run. Attempting to define a signal controlled by OpenEdition MVS will cause **sigdef** to fail. Note that if **oesigsetup** has not been called when **sigdef** is invoked, a default call to **oesigsetup** will be generated, as described in the **oesigsetup** function description.

The **name** argument enables you to rename the signal. **name** is the address of a null-terminated string with a maximum of five characters. The library appends these five characters to the **SIG** prefix to create the new name. For example, if the **name** argument specifies **EXT**, the signal name appears in messages and traces as **SIGEXT**.

The **dfl**, **intcntl**, **executive**, and **final** arguments indicate the addresses of functions that provide special processing for the signal. Each of these routines is described in detail later in this appendix. Coding a 0 for any of these arguments indicates that no function is supplied for that type of processing.

Note: If you specify 0 for the **dfl** argument, the signal is ignored when default handling is in effect.

RETURN VALUE

sigdef returns 0 if it completes successfully or nonzero if it cannot complete. Specifying an invalid signal number or one that has already been defined is the most common reason for failure.

SEE ALSO

Chapter 5, "Signal-Handling Functions," in *SAS/C Library Reference, Volume 1.*

Signal Generator Routine

The signal generator routine is the operating system exit that is invoked when the interrupt occurs; therefore, it is written in assembler language. (This routine can be written in C only if it is always called with C-compatible linkage, including the C use of register 12 and register 13. These conditions are unlikely to be met if this routine is an operating system exit.)

The signal generator routine raises the signal so that the library can detect it by calling the L$CZSYN routine (for synchronous signals) or the L$CZENQ routine (for asynchronous signals). These routines expect calls from assembler code and fully support them. The restrictions described in "Restrictions on Synchronous and Asynchronous Signals" on page 12-2 apply when you can issue calls to L$CZSYN. There are no restrictions on when you can call L$CZENQ. These routines are described in more detail in "SAS/C Library Routines for Adding New Signals" on page 12-8 .

The signal generator routine can build information that is passed to you when the user-defined handler calls **siginfo**. Refer to the SIGINFO field description in "Routine for Synchronous Signals: L$CZSYN" on page 12-9 for more information.

Default Routine

This routine simply defines what actions should occur when the user does not specify a signal handler for the signal. The default routine is usually coded in C. The initialization routine specifies the address of this routine as the second argument in the call to **sigdef**. If you do not define this routine, the default action is to ignore the signal. If you prefer to have the program ABEND by default, code this routine to call the **abort** function.

Note: Refer to the **siggen** or **abend** function to specify a particular ABEND code.

Interrupt Control Routine

The interrupt control routine enables you to communicate to the operating system that the handling for a user-defined signal has changed. The interrupt control routine is usually coded in assembler language. The initialization routine specifies the address of this routine as the third argument in the call to **sigdef**. This routine is called on the following occasions:

- when a call to **sigblock**, **sigsetmask**, or **sigprocmask** changes the mask for the signal.
- when a call to **signal** or **sigaction** requests default or ignore handling, or replaces default or ignore handling with a user-defined handler. If the call to **signal** merely replaces one user-defined handler function with another, the interrupt control routine may not be called.

The purpose of this routine is to improve performance by eliminating unnecessary processing. For example, if the operating system's default action for a signal is to ignore the signal, and the C program calls **signal** with a second argument of **SIG_IGN**, the interrupt control function can cause the operating system to ignore the signal when it occurs instead of calling an operating system exit. This saves the processing time required to transfer control to the default handler and produces the same results.

The linkage to the interrupt control routine is defined as follows:

```
int intcntl(int signum, int ignore, int default, int block,
            int context, struct sigaction *action)
```

The **signum** argument to the interrupt control routine specifies the signal number. The **ignore** and **default** arguments indicate how your program handles the signal. If **ignore** is 0, the program has either defined a signal handler or default handling is in

effect. If **ignore** is not 0, the signal is ignored. If **default** is 0, the program has either defined a signal handler or the signal is to be ignored. If **default** is not 0, default handling is in effect. (Both **ignore** and **default** are nonzero if default handling is in effect and the default action is to ignore the signal.) The **block** argument is nonzero if the signal is blocked or 0 if the signal is not blocked.

The **context** argument is an integer indicating the reason that the interrupt control routine was called. The possible values are **SC_ACTION**, **SC_PROCMASK**, **SC_DEF** and **SC_COPROCSWT**. (These are symbolic values defined in **<lcsignal.h>**.) **SC_ACTION** indicates a call as the result of a user call to **signal** or **sigaction**, **SC_PROCMASK** indicates a call as the result of a user call to **sigblock**, **sigsetmask** or **sigprocmask**, **SC_DEF** indicates a call as the result of the call to **sigdef**, and **SC_COPROCSWT** indicates a call as the result of a coprocess switch.

The **action** argument is meaningful only for calls with context **SC_ACTION** or **SC_DEF**. In these cases, **action** is a pointer to information about the current handling as defined by **sigaction**. The **sa_handler** field of the action should be ignored, as it may be different from the current handler. However, the **sa_mask** and **sa_flags** fields are guaranteed to be correct. In particular, this functionality allows the interrupt control routine to take special action based on the **SA_USRFLAG***n* flags settings.

The interrupt control routine should return a negative number to indicate an error. If a negative number is returned, the call to **signal** or **sigaction** that caused the interrupt control routine to be called returns **SIG_ERR**. A return code of 1, when the context is **SC_BLOCK**, indicates that the interrupt control routine takes no action for changes to blocking status. This enables the performance of signal processing to be improved by avoid calls to the interrupt control routine during **sigblock**, **sigsetmask**, or **sigprocmask** processing. Any other positive return code (or a 1 returned when the context is not **SC_PROCMASK**) is treated as a success.

Note: An interrupt control routine is rarely required for correct operation of signal code; this routine simply provides improved performance for signals that can be ignored or blocked by the operating system. There may be times, however, when the interrupt control routine actually increases overhead. For example, signals are blocked while I/O is performed, so the interrupt control routine is called several times for each I/O operation.

Executive Routine

The library calls the executive routine, if you code one, instead of calling the handler defined in your program. The executive routine is then expected to call the handler itself. If no executive routine is defined, the handler is called directly. Note that the executive routine is not called for a signal generated by **raise** or **siggen**; thus, the executive routine is entered only when a signal occurs naturally.

The executive routine is usually coded in C. If you code this routine in assembler language, use the **CENTRY** and **CEXIT** macros to avoid problems calling the user-defined handler. The initialization routine specifies the address of this routine as the fourth argument in the call to **sigdef**. The linkage to the executive routine is defined as follows:

```
void executive(int signum, char **infop, void (*handler)(int))
```

The **signum** argument is the number of the defined signal. The **infop** pointer addresses the pointer that the signal handler in the program can access by a call to **siginfo**. The executive routine may modify the information addressed by this pointer. The **handler** argument addresses the user-defined handler or contains 0 if the signal is to be ignored. This address can be used to call the user-defined handler from the executive routine.

The executive routine can be used to monitor or prevent certain handler activity. For instance, you can use **blkjmp** to prevent successful use of **longjmp** by the handler, or you can refuse to allow normal return from the handler by calling **abort** if the handler returns. It is assumed that the executive routine will call the handler using the normal handler linkage, but this cannot be enforced.

Final Routine

The final routine is called on program termination to allow signal-handling to be cancelled. Normally, a final routine is provided to inform the operating system that handling of the interrupt associated with the signal is no longer required.

The final routine is usually coded in assembler language. The initialization routine specifies the address of this routine as the fifth argument in the call to **sigdef**. The linkage to the final routine is defined as follows:

```
void final(int signum)
```

The **signum** argument is the number of the signal whose handling will be terminated.

The final routine is called after all files have been closed by the library. Therefore, this routine is not permitted to use I/O.

Jump Intercept Routine

The jump intercept routine is invoked if a signal handler for a synchronous signal issues a **longjmp**. This routine must be coded in assembler language. The jump intercept routine can inform the operating system that handling of the interrupt is complete but that control should not return to the point of interrupt. This is sometimes called *clearing the interrupt*. The jump intercept routine may need to be called before performing the **longjmp** because the **longjmp** routine prevents return to the signal generation routine from the handler and, therefore, also prevents normal return to the operating system.

The jump intercept routine is rarely coded because it is frequently impossible to correctly clear the interrupt. If you expect a user-defined handler to call **longjmp**, you can disallow **longjmp** or define the signal as asynchronous. The jump intercept routine should not attempt to prevent a **longjmp**. If you want to disallow jumps, use the executive routine to block them.

The jump intercept routine is not included as an argument in the call to **sigdef**. To indicate that you want to provide this routine, set the JUMPINT field of the **ZSYNARGS** DSECT to the address of the jump intercept routine. The jump intercept routine is called using standard MVS linkage. Register 1 addresses the save area where registers have been saved (except for register 14 and register 15, which are not available).

SAS/C Library Routines for Adding New Signals

The library provides two routines (L$CZSYN and L$CZENQ) that can communicate to the library that a user-defined signal has occurred. These routines should be called from the signal generating routine, which is normally written in assembler language. The library expects calls to these routines from assembler language and fully supports them.

Routine for Synchronous Signals: L$CZSYN

When you call L$CZSYN to inform the library of a synchronous signal, register 12 must address the CRAB, and register 13 must address the current C DSA. Register 1 addresses an argument list described by the following DSECT:

```
ZSYNARGS  DSECT

SIGNUM    DS F   signal number
SIGINFO   DS A   address of associated information
ABCODE    DS A   pointer to associated ABEND code (null-terminated),
                 if any
SIGNAME   DS A   pointer to five-character signal name
SIGLOC    DS A   pointer to the interrupted instruction
JUMPINT   DS A   address of a jump intercept routine, if needed
```

You must provide all of the information for these fields. The fields are described as follows:

SIGNUM contains the number of the signal.

SIGINFO contains a pointer to the value that will be made available to the signal handler with the **siginfo** function. Refer to Chapter 5, "Signal-Handling Functions," in *SAS/C Library Reference, Volume 1* for more information on **siginfo**.

The signal-generating routine builds this information and stores the address in this field. Then the address is passed to the executive routine for the signal, if any. The executive routine may want to modify or make a copy of this information. For example, you might pass L$CZSYN the address of a system control block. The executive routine can make a copy of it to pass to the user-defined handler. This stops the user from attempting to modify the control block.

ABCODE can be 0 or it can contain the address of a string that contains an ABEND code associated with the signal. The ABEND code should be null-terminated and should begin with a **'U'** if it is a user ABEND rather than a system ABEND. For example, if you define a signal associated with exceeding a CPU-time quota, you probably would define ABCODE as **''322''** because that is the ABEND code normally produced by this condition.

SIGNAME contains the address of the five-character string passed as the last argument in the call to **sigdef**. These five characters are appended to **SIG** to form a new name that replaces the standard name, **SIGUSR1–8**.

SIGLOC should contain the address in the C program where processing was interrupted. This address is used in tracebacks if an ABEND occurs or the debugger **where** command is used. If you cannot provide this information, set this field to **NULL**.

JUMPINT can provide a jump intercept routine. The normal use of a jump intercept routine is to inform the operating system that the interrupt has been handled. Set this field to 0 if you have not coded a jump intercept routine. Refer to the "Jump Intercept Routine" on page 12-8 for more information on this routine.

L$CZSYN returns either a 0 or a 4 in register 15 unless handling of the signal is terminated by a **longjmp**, in which case no return occurs. If L$CZSYN returns a 4, the

signal cannot be processed because one or more of the restrictions on the timing of synchronous signals is violated. Refer to "Restrictions on Synchronous and Asynchronous Signals" on page 12-2 for more information. If L$CZSYN returns a 0, the signal is accepted, and a user-defined signal handler is called and returned. In either case, the signal generation routine should return to the operating system.

Routine for Asynchronous Signals: L$CZENQ

The L$CZENQ routine can be called at any time to inform the library that an asynchronous signal has occurred. Unlike L$CZSYN, L$CZENQ does not cause a handler to be immediately called. Instead, L$CZENQ adds the signal to an internal queue of pending signals. No handler is called until the signal can be discovered; discovery occurs when a function is called or returns and the signal is not blocked. L$CZENQ can be called in situations where normal C code cannot be executed, such as under an SRB or a subtask TCB in MVS, or from a CMS interrupt handler. L$CZENQ is reliable whether or not hardware interrupts are disabled at the time it is called. Recursive interrupts and simultaneous interrupts in a multitasking or multiprocessing environment are supported.

The address of L$CZENQ is located at offset X'1F4' (decimal 500) from the start of the CRAB. Linkage to L$CZENQ should be effected using this CRAB field, not a V-type address constant. Using a V-type constant works only if the calling routine is linked with the main load module of the application program.

The following shows the use of various registers by L$CZENQ:

Register	Use
1	addresses parameter list ZENQARGS
2–6	work registers; save contents before calling L$CZENQ
12	must address CRAB
13	ignored; no registers saved by L$CZENQ
14	contains return address
15	contains address of L$CZENQ

The parameter list addressed by register 1 is described by the following DSECT:

```
ZENQARGS  DSECT

SIGNUM    DS F   signal number
SIGINFO   DS A   address of associated information
ABCODE    DS A   pointer to associated ABEND code (null-terminated),
                 if any
SIGNAME   DS A   pointer to five-character signal name
SIGELEM   DS A   address of an interrupt element
ECBPP     DS A   address of a word in which to store an ECB address
```

The first four fields have the same meaning as the corresponding fields in L$CZSYN. The SIGELEM and ECBPP fields should be used as follows:

SIGELEM is required. It must address a 24-byte area of storage that can be used as an interrupt element by L$CZENQ. Under MVS, this element must be allocated by GETMAIN from subpool 1. Under CMS, this element must be allocated through use of CMSSTOR (or DMSFREE under 370 mode CMS). The element must be accessible using a 24-bit address. The element is freed by the run-time library after processing of this signal is complete.

ECBPP can address a word of memory or can contain 0s. If ECBPP is not 0, and the program is executing **pause**, **sigpause**, **sigsuspend**, or **sleep** at the time of the call to L$CZENQ, the address of the Event Control Block (ECG) used by **pause**, **sigpause**, **sigsuspend**, and **sleep** is stored in the word addressed by ECBPP. If ECBPP is 0, the ECB address is not stored; instead, the ECB is posted using SVC 2. Therefore, if you call L$CZENQ in a situation where SVC's cannot be issued (such as from an SRB routine or I/O appendage), you must provide an ECBPP value. Note that **pause**, **sigpause**, **sigsuspend**, and **sleep** do not complete until this ECB is posted. For this reason, in such cases you would normally call a branch entry to POST to awaken the C program.

L$CZENQ does not have a return code because L$CZENQ cannot fail without causing abnormal program termination.

13 Getting Environmental Information

13-1 Introduction
13-1 The L$UENVR Routine
13-1 L$UENVR Source Code
13-1 Environment Descriptor Block
13-3 Caution
13-3 Function Descriptions

Introduction

Most C programs run in a standard operating environment that requires no modification to the SAS/C Library. However, in a nonstandard environment, the L$UENVER routine must be modified to provide information about the special operating environment.

The L$UENVR Routine

L$UENVR is a routine called by the SAS/C Library as part of initialization every time the **`main`** function of a C program executes. L$UENVR returns information about the operating system and the current environment.

This description of L$UENVR has two objectives:

- It provides general information about L$UENVR for systems programmers who may need to modify L$UENVR to run C programs in special environments.
- It describes several library functions associated with L$UENVR that are available to any programmer. These functions (**`envlevel`**, **`envname`**, **`intractv`**, **`syslevel`**, and **`sysname`**) return operating system and environment information.

L$UENVR Source Code

A sample version of L$UENVR source code is provided on the installation tape (in member L$UENVR). * This sample returns information about the standard systems (MVS and CMS) under which the compiler runs. The sample is included so that L$UENVR can be modified by a site to indicate special environments. Comments in the code describe how to make any necessary modifications.

Your site normally does not need to change L$UENVR unless you are running programs in a nonstandard environment. For example, changes in the L$UENVR source code are needed to run C programs under a system (such as ROSCOE) that is different from the systems for which the compiler was developed. Note that the operation of the library in such special environments is not guaranteed, but if the environment resembles a normal MVS or CMS environment closely enough, execution is possible.

Environment Descriptor Block

The L$UENVR routine stores information about the operating system in an environment descriptor block. L$UENVR is called using standard IBM linkage. Register 1 addresses an area in which the environment descriptor block will be built. (This block is mapped by the ENVDB DSECT, which is present in the sample macro

* See your SAS Software Representative for SAS/C software products for the location of this member.

library.) Fifty-six bytes that can be used for work space follow the standard 72-byte save area addressed by register 13.

The environment descriptor block is 48 bytes and contains the information shown in the following table.

Table 13.1
Environment Descriptor Block

Offset (decimal)	Length (bytes)	Description
0	2	halfword length of the block (48) (decimal)
2	8	environment name (blank padded), for example, MVS/SP, VM/SP
10	1	environment version number (integer)
11	1	environment release number (integer)
12	1	environment modification level (integer)
13	1	submodification level, if appropriate
14	8	subenvironment name, if applicable; for example, TSO—a subenvironment of MVS
22	1	subenvironment version number (integer)
23	1	subenvironment release number (integer)
24	1	subenvironment modification level (integer)
25	1	subenvironment submodification level
26	1	environment number (see ENVDB DSECT for codes)
27	1	supported function flags. These flags are set by L$UENVR to show whether specific functions are available.
28	1	XA support flags `X'80'` = XA architecture, `X'40'` = PSW in XA format
29	1	time sharing flags `X'80'` = program executing interactively
30	1	system function flags: `X'80'` = dynamic allocation supported `X'40'` = SPIE or ESPIE supported `X'20'` = ESTAE or ABNEXIT supported
31	18	reserved

Any undefined or meaningless items in the environment descriptor block should be stored as `0`s. If you modify L$UENVR, a new set of information is created for the special environment in which compiled programs will run.

Note: If you modify L$UENVR, do not change the information in the descriptor block for the standard systems (MVS and CMS) because the library may malfunction.

Caution The information may be unavailable or incorrect unless the L$UENVR source code is examined and changed as necessary by a systems programmer familiar with the local operating systems.

Local operating system conventions or modifications may cause these functions to return incorrect or misleading values. Contact your SAS Software Representative for C Compiler products to confirm that L$UENVR provides the information required.

Function Descriptions

Descriptions of each system interface function follow. Each description includes a synopsis, a description, discussions of return values and portability issues, and an example. Also, errors, cautions, diagnostics, implementation details, and usage notes are included if appropriate.

envlevel Get Subenvironment Information

SYNOPSIS

```
#include <lclib.h>

char *envlevel(void);
```

DESCRIPTION

envlevel gets the subenvironment version number, release number, modification level, and submodification level. This information is stored in bytes 22 through 25 of the environment descriptor block. If no subenvironment is active, **envlevel** contains the same information as **syslevel**.

A null character is stored for any piece of information that is not meaningful on the current system.

RETURN VALUE

envlevel returns a pointer to a character sequence containing four pieces of information: subenvironment version number, release number, modification level, and submodification level.

The type of information returned by **envlevel** is site-dependent. For example, under TSO/E, the release number field contains the release of TSO/E. Under CMS, it typically contains the CMS release number. None of the other bytes are meaningful.

CAUTION

See “Caution” under “The L$UENVR Routine” on page 13-1 .

EXAMPLE

This example formats and prints the **envname/envlevel** and **sysname/syslevel** information.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <lclib.h>

main()
{
   char  *sysptr, *envptr, sysstr[16], envstr[16];

   envptr = envlevel();
   sprintf(envstr, "%02x %02x %02x %02x",
           *envptr, *(envptr + 1), *(envptr + 2), *(envptr + 3));
   printf("Envname/Envlevel  =  %-8s %s\n", envname(), envstr);

   sysptr = syslevel();
   sprintf(sysstr, "%02x %02x %02x %02x",
           *sysptr, *(sysptr + 1), *(sysptr + 2), *(sysptr + 3));
   printf("Sysname/Syslevel  =  %-8s %s\n", sysname(), sysstr);
}
```

envname Get Subenvironment Name

SYNOPSIS

```
#include <lclib.h>

char *envname(void);
```

DESCRIPTION

envname gets the subenvironment name stored in bytes 14 through 21 of the environment descriptor block.

RETURN VALUE

envname returns a pointer to the string containing the subenvironment name. Under MVS, if the program is running under TSO, **envname** returns either **''TSO''** or **''TSO/E''**. Under CMS, **envname** returns **''CMS''**. For an OpenEdition child process, or a program called with **exec**-linkage, **envname** returns **''OpenMVS''**, even if invoked under TSO.

If no special subenvironment is active, **envname** returns the same information as **sysname**.

CAUTION

See "Caution" under "The L$UENVR Routine" on page 13-1 .

EXAMPLE

```
#include <lclib.h>
#include <stdlib.h>
#include <lcstring.h>

   if (memcmp(envname(), "TSO", 3) == 0)
      system("TSO: DELETE TEMP.FILE");
   else if (memcmp(envname(),"CMS", 3) == 0)
      system("CMS: ERASE TEMP FILE A");
   else if (memcmp(envname(), "OpenMVS", 7) == 0)
      system("sh: rm temp.file");
   .
   .
   .
```

envname is also illustrated in the example for **envlevel**.

intractv Indicate Interactive Execution

SYNOPSIS

```
#include <lclib.h>

int intractv(void);
```

DESCRIPTION

intractv indicates whether a program is executing interactively. The time-sharing flag is stored in byte 29 of the environment descriptor block.

Execution under MVS-batch, including execution of the TSO terminal monitor program, is not interactive. TSO executes programs interactively. CMS execution is normally interactive. However, if there is no interactive console associated with the executing program, a C program under CMS is considered noninteractive. Consult the systems programmer at your local site to determine under what conditions a program is considered noninteractive for CMS.

RETURN VALUE

Except for OpenEdition programs called with **exec**-linkage, **intractv** returns a nonzero integer if a program is executing interactively; otherwise, it returns **0**.

For a program called with **exec**-linkage, **intractv** returns whether or not a TSO terminal is accessible. Therefore, in most cases of **exec**-linkage, **intractv** returns **0**; however, for a program invoked via the **oeattach** or **oeattache** function under TSO, interactive returns nonzero.

CAUTION

See “Caution” under “The L$UENVR Routine” on page 13-1 .

EXAMPLE

```
#include <lclib.h>
#include <stdio.h>
#include <string.h>

FILE *in_file;
char *sys_file;

      /* Issue prompt only if running interactively.          */
   if (intractv()) {
      printf("Enter data:");
      fflush(stdout);

      /* Perform other functions to read data in interactively.*/
      .
      .
      .
   }
```

intractv Indicate Interactive Execution

(continued)

```
   /* Otherwise, obtain input from DATA file.              */
else {
   if (memcmp(sysname(), "CMS", 3) == 0)
      sys_file = "DATA FILE A1";
   else
      sys_file = "SYSIN";
   in_file = fopen(sys_file, "r");
}
```

syslevel Get Operating System Information

SYNOPSIS

```
#include <lclib.h>

char *syslevel(void);
```

DESCRIPTION

syslevel gets the operating system version number, release number, modification level, and submodification level stored in bytes 10 through 13 of the environment descriptor block.

A null character is stored for any piece of information that is not meaningful on the current system.

RETURN VALUE

syslevel returns a pointer to the character sequence containing the operating system version number, release number, modification level, and submodification level.

The type of information returned by **syslevel** is site-dependent. An example of typical information provided under MVS is the version and release number of MVS. Under CMS, only the release number is meaningful; this is the release of the VM control program (CP).

CAUTION

See “Caution” under “The L$UENVR Routine” on page 13-1 .

EXAMPLE

syslevel is illustrated in the example for **envlevel**.

sysname Get Operating System Name

SYNOPSIS

```
#include <lclib.h>

char *sysname(void);
```

DESCRIPTION

sysname gets the operating system name from bytes 2 through 9 in the environment descriptor block.

RETURN VALUE

sysname returns a pointer to a string containing the operating system name.

Under MVS, the string returned by **sysname** is **''VS1''**, **''MVS''**, or **''MVS/SP''**. Under CMS, **''VM/SP''**, **''VM/XA SP''**, **''VM/HPO''**, **''VM/XA''**, or **''VM/ESA''** is returned.

CAUTION

See “Caution” under “The L$UENVR Routine” on page 13-1 .

EXAMPLE

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

char *pathname;

   if (strcmp(sysname(), "VM/SP") == 0)
      pathname = "PRINTER";
   else
      pathname = "SYSPRINT";
   .
   .
   .
```

sysname is also illustrated in the example for **envlevel**.

RELATED FUNCTIONS

uname

Part 2

SAS/C® Socket Library for TCP/IP

Chapters

14 The TCP/IP Protocol Suite
15 The BSD UNIX Socket Library
16 Porting UNIX Socket Applications to the SAS/C® Environment
17 Network Administration
18 Socket Function Reference

14 The TCP/IP Protocol Suite

14-1 Overview of TCP/IP
14-2 Internet Protocol (IP)
14-2 User Datagram Protocol (UDP)
14-3 Transmission Control Protocol (TCP)
14-3 Domain Name System (DNS)
14-4 MVS and CMS TCP/IP Implementation

Overview of TCP/IP

Networking has become a fundamental feature of most computer applications. (TCP/IP), the protocol suite used by the Defense Advanced Research Projects Agency (DARPA) Internet, is one of the most commonly used network protocols. The DARPA Internet is a collection of networks and gateways that function as a single network. The Internet extends all over the globe and consists of over 100,000 computers.

Finding a way to connect existing computer networks of diverse types was a primary goal in the design of TCP/IP. Achieving of this goal has made TCP/IP particularly successful in the networking of computers that run different operating systems and that are manufactured by different vendors. TCP/IP is also designed to accommodate a very large number of host computers and local networks. A site or organization that uses TCP/IP need not be connected to the Internet, but most sites and organizations are connected.

The open, nonproprietary nature of TCP/IP and its global scope have made it popular among users of the UNIX operating system. Standards for writing communications programs in C have also become widespread. The two most common standards are the BSD UNIX Socket Library interface and the UNIX System V Transport Layer Interface (TLI).

The SAS/C Library currently implements the BSD UNIX Socket Library interface because it is somewhat more common than TLI, and it has better support from the underlying communications software on MVS and CMS systems. The socket library is integrated with SAS/C support for UNIX file I/O to provide the same type of integration between file and network I/O that is available on BSD UNIX systems.

TCP/IP is now a base for higher level protocols that support many popular networking applications. Some of these protocols are:

TELNET	a remote terminal connection service that supports remote login.
FTP	a File Transfer Protocol that transfers files from one machine to another.
X11	a graphical user interface that can operate in a network environment. This protocol is not limited to TCP/IP.
NFS	a Network File System that allows cooperating computers to access one another's file systems as though they were local. This protocol is not limited to TCP/IP.

The multivendor capabilities of these protocols and the applications that use them have made them particularly successful.

Internet Protocol (IP)

The Internet Protocol (IP) integrates different physical or proprietary networks into a unified logical network known as the Internet.

An IP address is a 32-bit number that specifies both the number for the individual physical network and the number for a given host computer on that network. The term *host computer* can apply to any end-user computer system that connects to a network. The size of a host can range from an X-terminal or a PC to a large mainframe. Among all the organizations connected to the Internet, the address of each host computer is unique. The address is often written in dotted decimal notation. *Dotted decimal notation* is the decimal value of each byte (often referred to as an octet in the literature on TCP/IP) separated by a period. For example,

192.22.31.05

is the dotted decimal notation for a machine whose 32-bit address is

0xC0161f05

At the IP protocol layer, host computers cannot be referenced by name. Refer to "Domain Name System (DNS)" on page 14-3 for an explanation of name referencing for host computers. All network communication uses IP addresses. The IP layer routes packets of data to their destinations, which may be many physical hops away from the source of the message. A *physical hop* is a gateway through which the data must pass. The IP layer does not guarantee that a packet will reach its destination, nor does it provide error checking for the data. The IP layer does not provide flow control or any lasting association (connection) between sender and receiver. A higher layer of the protocol, such as the User Datagram Protocol (UDP) or TCP, must provide all these services.

User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) provides the lowest level of service that can be used conveniently by network application programs. UDP is most often used when applications implement their own networking protocol and thus require little intervention from the TCP/IP software.

In addition to the communication capabilities of IP, UDP adds checksums for the application data and protocol ports to help distinguish among the different processes that are communicating between sending and receiving machines. A *checksum* detects errors in the transfer of a packet from one machine to another. A *protocol port* is an abstraction used to distinguish between multiple destinations within a single host.

A *datagram* is a basic unit of information transferred across a network. UDP does not guarantee that datagrams reach their destination, nor does it ensure that the datagrams are received in the order in which they are sent. Because UDP does not use connections or sessions, it is called a *connectionless protocol.*

UDP ports are two-byte integers that specify a particular service or program within a host computer. For example, port 13 is generally used by programs that query the date and time maintained by a particular host. A client-server relationship is usually defined in a UDP transaction. The server waits for messages (listens) at a predefined port. When it receives a datagram from a new client, the server knows where to respond because the datagram contained both the sender's IP address and its port number.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) provides a higher level of service. TCP establishes connections between the sender and receiver by using IP addresses and port numbers. It then simulates a bidirectional, continuous communications service. TCP provides reliability of transmission, division of data into packets (without the knowledge of the user program), ordered transmission of packets, simultaneous transfer in both directions, and buffering of data. The format of TCP data is analogous to that of a file in the UNIX operating system. Data are sent or received in a continuous stream of bytes. The data have no record (message) boundaries or any other structure except that agreed to by the connected applications.

Domain Name System (DNS)

The previously discussed protocols do not use any concept of a host name. These protocols always use 32-bit IP addresses to locate source and destination hosts. Of course, no one wants to specify a remote computer using an address such as 192.22.31.05. The Domain Name System (DNS) maps IP addresses to alphabetic names. One of the most important features of DNS is distributed management.

Each organization has the ability to control names within its own domain. Domains are arranged in a hierarchy. For example, the XYZ Company, Inc., may have names all ending in the following:

.xyz.com

com the final section of the name, is a higher-level domain used to group commercial organizations.

xyz the second section of the name, is the designated name of the organization.

The names could be further divided into several groups such as the following:

unx

vm

dev

For example,

abcvm.vm.xyz.com

might be the primary VM system at the XYZ Company, Inc. DNS enables you to use a File Transfer Program command such as

ftp abcvm.vm.xyz.com

instead of

ftp 123.45.67.89

when transferring a file to this VM system.

Although it is possible to locate the mapping of host addresses to host names in a file (for example, **`/etc/hosts`** on UNIX), DNS is more versatile than a system that maps addresses to names in a file. Under a system that maps names to addresses, the file containing the mapped names and addresses: must be replicated on every host, does not have the capacity to contain the mappings for all computers on a system as large as the Internet, and cannot be updated on a real-time basis.

DNS uses server processes called *name servers* to stay current with the names assigned within a particular domain. The network administrator provides the name servers with configuration files. Each configuration file contains the mapping for the domain that it controls. Name servers in a particular domain can refer to the addresses of name servers for higher- and lower-level domains if the configuration files that they control do not contain a particular name or address.

Name servers typically run on only a few machines in an organization. Programs can use a set of routines, known as the *resolver*, to query their organization's name server. The resolver routines are associated with the application and provide all the message formatting and TCP or UDP communications logic necessary to talk to their organization's name server.

DNS is general enough to allow distributed management of other types of information, such as mailbox locations, and it does not require any correspondence between domains and IP addresses or physical network connections.

MVS and CMS TCP/IP Implementation

The SAS/C Socket Library has an open architecture that permits using TCP/IP products from different vendors. If a vendor provides the appropriate SAS/C transient library module, existing socket programs can communicate using the TCP/IP implementation specified during configuration of the system. A program compiled and linked with the SAS/C Socket Library at one site can be distributed to sites that are running different TCP/IP implementations. In addition, any site can change TCP/IP vendors without recompiling or relinking its existing SAS/C applications.

With Release 6.00, the SAS/C Library supports both integrated and non-integrated sockets. With integrated sockets, the TCP/IP sockets are integrated with OpenEdition support instead of being a direct run-time library interface to the TCP/IP software implemented only in the run-time library.

With non-integrated sockets, the SAS/C Socket Library relies on an underlying layer of TCP/IP communications software, such as IBM TCP/IP Version 2, or higher, for VM and MVS. TCP/IP communications software handles the actual communications. The SAS/C Library adds a higher level of UNIX compatibility, as well as integration with the SAS/C run-time environment.

15 The BSD UNIX Socket Library

15-1 Introduction
15-2 Overview of the BSD UNIX Socket Library
15-3 Header Files
15-3 <sys/types.h>
15-4 <sys/uio.h>
15-4 <errno.h>
15-7 <sys/ioctl.h>
15-7 <fcntl.h>
15-7 <sys/socket.h>
15-8 <netdb.h>
15-9 <netinet/in.h>
15-10 <netinet/in_systm.h>
15-10 <netinet/ip_icmp.h>
15-11 <netinet/ip.h>
15-11 <netinet/udp.h>
15-11 <arpa/inet.h>
15-11 <arpa/nameser.h>
15-11 <resolv.h>
15-12 <net/if.h>
15-12 <strings.h>
15-12 Socket Functions
15-13 Addresses and Network Information
15-14 Data Conversion
15-15 Choosing a Socket Implementation
15-15 Creating a Socket
15-15 Binding a Socket
15-15 Listening for a Connection
15-16 Connecting and Passing a Socket
15-16 Accepting a Connection
15-16 Sending and Receiving Data Through a Socket
15-16 Closing the Connection

Introduction

This chapter contains a discussion of the Berkeley Software Distribution (BSD) UNIX Socket Library and the application programming interface for writing TCP/IP applications using the SAS/C Compiler. It also describes the purpose of each header file. Finally, it describes the basic mechanics of communicating through sockets, along with a summary of the socket functions provided with the SAS/C Compiler.

Overview of the BSD UNIX Socket Library

BSD UNIX communications programming is based on the original UNIX framework, with some additions and elaborations to take into account the greater complexity of interprocess communications. In traditional UNIX file I/O, an application issues an **`open`** call, which returns a file descriptor (a small integer) bound to a particular file or device. The application then issues a **`read`** or **`write`** call that causes the transfer of stream data. At the end of communications, the application issues a **`close`** call to terminate the interaction.

Because interprocess communication often takes place over a network, the BSD UNIX Socket Library takes into account the numerous variables of network I/O, such as network protocols, in addition to the semantics of the UNIX file system.

Briefly stated, a *socket* is an end point for interprocess communication, in this case, over a network running TCP/IP. The socket interface can support a number of underlying transport mechanisms. Ideally, a program written with socket calls can be used with different network architectures and different local interprocess communication facilities with few or no changes. The SAS/C Compiler supports TCP/IP and the AF_INET Internet addressing family.

Sockets can simultaneously transmit and receive data from another process, using semantics that depend on the type of socket. There are three types of sockets: stream, datagram, and raw, each of which represents a different type of communications service.

Stream sockets provide reliable, connection-based communications. In connection-based communications, the two processes must establish a logical connection with each other. A stream of bytes is then sent without errors or duplication and is received in the order in which it was sent. Stream sockets correspond to the TCP protocol in TCP/IP.

Datagram sockets communicate via discrete messages, called datagrams, which are sent as packets. Datagram sockets are connectionless; that is, the communicating processes do not have a logical connection with each other. The delivery of their data is unreliable. The datagrams can be lost or duplicated, or they may not arrive in the order in which they were sent. Datagram sockets correspond to the UDP protocol in TCP/IP.

Raw sockets provide direct access to the lower-layer protocols, for example, IP and the Internet Control Message Protocol (ICMP).

If you are communicating with an existing application, you must use the same protocols as that application. When symbols defined in the header files are passed as parameters to the socket functions, you can change options and variables, such as whether your communications are connection-oriented or connectionless and what network names you want to send to or receive from.

In network operations, an application can specify a different destination each time it uses a socket. It is also possible to send datagrams of varying length, in addition to stream data. The socket interface enables you to create both servers that await connections and clients that initiate the connections using services in addition to **`open`**, **`read`**, **`write`**, and **`close`**.

Another factor that makes communicating over a network more complex than traditional UNIX I/O is the problem of ascertaining the correct addresses for clients and servers, and of converting information from one format to another as it travels between different machines on different networks in the Internet. The socket library provides Internet addressing utility functions and network information functions to resolve these issues.

Header Files

The SAS/C Socket Library provides header files to enable you to program with socket functions. A list of the header files, accompanied by a brief description of each one and an explanation of its structures, follows. Refer to "Header Filenames" on page 16-3 for a discussion of header file naming conventions.

The header files in this section are listed in the following order:

1. **<sys/types.h>**
2. **<sys/uio.h>**
3. **<errno.h>**
4. **<sys/ioctl.h>**
5. **<fcntl.h>**
6. **<sys/socket.h>**
7. **<netdb.h>**
8. **<netinet/in_systm.h>**
9. **<netinet/ip_icmp.h>**
10. **<netinet/udp.h>**
11. **<netinet/ip.h>**
12. **<netinet/in.h>**
13. **<arpa/inet.h>**
14. **<arpa/nameser.h>**
15. **<resolv.h>**
16. **<net/if.h>**
17. **<strings.h>**

In many cases the contents of one header file depend on the prior inclusion of another header file. The description for each header file lists any other header files on which the header file may depend. Failure to adhere to these ordering dependencies usually results in compilation errors. The order that the header files follow in this chapter reflects the dependencies between header files.

Note: Sockets are not defined by the POSIX.1 and POSIX.1a standards. Therefore, if your program uses sockets, you should not define the symbol **_POSIX_SOURCE**, as this will exclude socket-related types from standard POSIX header files such as **<sys/types.h>**.

<sys/types.h>

This header file contains definitions to allow for the porting of BSD programs.

<sys/types.h> must usually be included before other socket-related header files; refer to the individual header file descriptions that follow for the specific dependency.

This header file declares the following **typedef** names for use as abbreviations for three commonly used types:

```
typedef unsigned char;
u_char;

typedef unsigned short;
u_short;

typedef unsigned long;
u_long;.
```

The following **typedef** name is commonly used for buffer pointers:

```
typedef char * caddr_t;
```

This header file also defines the **FD_SET**, **FD_ZERO**, **FD_CLR**, **FD_ISSET**, and **FD_SETSIZE** macros used by **select** to manipulate socket descriptors. Refer to Chapter 18, "Socket Function Reference" on page 18-1 for a description of these macros.

timeval

This structure is used by the **select** call to set the amount of time for a user process to wait.

```
struct timeval {
   long tv_sec;     /* seconds      */
   long tv_usec;    /* microseconds */
};
```

For more information on the inclusion of this structure in the **<sys/types.h>** header file, refer to Chapter 16, "Porting UNIX Socket Applications to the SAS/C Environment" on page 16-1

<sys/uio.h>

The structure in the **<sys/uio.h>** header file is described in the following section. **<sys/types.h>** must be included before this header file.

iovec

This structure is used by the **readv**, **writev**, **sendmsg**, and **recvmsg** calls. An array of **iovec** structures describes the pieces of a noncontiguous buffer.

```
struct iovec {
   void * iov_base;
   int    iov_len;
};
```

<errno.h>

This header file contains definitions for the macro identifiers that name system error status conditions. When a SAS/C Library function sets an error status by assigning a nonzero value to **errno**, the calling program can check for a particular value by using the name defined in **<errno.h>**. The following table lists all the **errno** values normally associated with the socket library functions:

Errno Value	perror message	Explanation
EACCES	permission denied	The program does not have access to this socket.
EADDRINUSE	socket address is already being used	The given address is already in use.
EADDRNOTAVAIL	socket address not usable	The given address is not available on the local host.
EAFNOSUPPORT	unsupported socket addressing family	The addressing family is not supported or is not consistent with socket type. The SAS/C Library supports only **AF_INET** (and **AF_UNIX** if integrated sockets are used).

continued

Errno Value	perror message	Explanation
EALREADY	previous connection not yet completed	The socket is marked for non-blocking I/O, and an earlier **connect** call has not yet completed.
EBADF	file or socket not open or unsuitable	The descriptor value passed does not correspond to an open socket or file.
ECONNABORTED	connection aborted by local network software	The local communications software aborted the connection.
ECONNREFUSED	destination host refused socket connection	The destination host refused the socket connection.
ECONNRESET	connection reset by peer	The peer process has reset the connection.
EDESTADDRREQ	socket operation requires destination address	Supply a destination address for this socket.
EHOSTDOWN	destination host is down	The destination host is down.
EHOSTUNREACH	destination host is unreachable	The destination host is unreachable.
EINPROGRESS	socket connection in progress	The connection has begun, but control is returned so that the call will not block. The connection is complete when the **select** call indicates that the socket is ready for writing. This is not an error.
EISCONN	socket is already connected	The program called **connect** on a connected socket.
EMSGSIZE	message too large for datagram socket	A datagram socket could not accommodate a message as large as this one.
ENETDOWN	local host's network down or inaccessible	The program cannot talk to the networking software on this local machine, or the host's network is down.
ENETRESET	remote host dropped network communications	The remote host is not communicating over the network at this time.

continued

Errno Value	perror message	Explanation
ENETUNREACH	destination network is unreachable	This host cannot find a route to the destination network.
ENOBUFS	insufficient buffers in network software	The operating system did not have enough memory to perform the requested operation.
ENOPROTOOPT	option not supported for protocol type	The socket option or option level is invalid.
ENOTCONN	socket is not connected	The socket is not connected.
ENOTSOCK	file descriptor not associated with a socket	The given file descriptor is either assigned to a file or is completely unassigned.
EOPNOTSUPP	operation not supported on socket	The call does not support this type of socket.
EPFNOSUPPORT	unsupported socket protocol family	The given protocol family is unknown or unsupported by the TCP/IP network software.
EPIPE	broken pipe or socket connection	The peer process closed the socket before your process was finished with it.
EPROTONOSUPPORT	unsupported socket protocol	The given protocol is unknown or not supported by the TCP/IP network software.
EPROTOTYPE	protocol inconsistent with socket type	When calling the socket, the protocol was not `0` and not consistent with the socket type.
ESHUTDOWN	connection has been shut down	The connection has been shut down.
ESOCKTNOSUPPORT	socket type not allowed	The program specified a socket type that is not supported by the TCP/IP network software.
ESYS	operating system interface failure	The underlying operating system software or the TCP/IP software returned an abnormal failure condition. Contact TCP/IP vendor.
ETIMEDOUT	socket connection attempt timed out	The destination host did not respond to a connection request.
EWOULDBLOCK	socket operation would block	The socket is marked for non-blocking I/O, and the call would have been blocked. This is not an error.

Note: Socket library functions may occasionally set **errno** values not shown in the above table. In particular, when integrated sockets are used, OpenEdition defined **errno** values may be stored to indicate conditions specific to the OpenEdition implementation. Common SAS/C **errno** values are listed in Chapter 1, "Introduction to the SAS/C Library," in *SAS/C Library Reference, Volume 1.* Also refer to "POSIX and OpenEdition Error Numbers" on page 19-4 for OpenEdition related **errno** values. For a complete listing of all **errno** values, see the *SAS/C Compiler and Library Quick Reference Guide.*

<sys/ioctl.h> This header file contains definitions for the symbols required by the **ioctl** function, as well as the declaration for **ioctl**.

<fcntl.h> This header file contains definitions for the constants associated with the **fcntl** function, as well as declarations for UNIX style I/O functions. Failure to include the **<sys/uio.h>** header file before this header file may result in a warning message if **readv** or **writev** is called.

<sys/socket.h> This header file contains macro definitions related to the creation of sockets, for example, the type of socket (stream, datagram, or raw), the options supported, and the address family. (**AF_UNIX** is supported if integrated sockets are used.) The SAS/C Compiler only supports the TCP/IP and the **AF_INET** Internet address family. The **<sys/socket.h>** header file contains declarations for most of the functions that operate on sockets. You must include the **<sys/types.h>** header file before this header file. The structures in the **<sys/socket.h>** header file are described in the following sections.

linger

This structure is used for manipulating the amount of time that a socket waits for the transmission of unsent messages after the initiation of the **close** call.

```
struct linger {
   int     l_onoff;          /* option on/off                    */
   int     l_linger;         /* linger time, in seconds          */
};
```

sockaddr

This is a generic socket address structure. Because different underlying transport mechanisms address peer processes in different ways, the socket address format may vary.

```
struct sockaddr {
   u_short sa_family;        /* address family                   */
   char    sa_data[14] ;     /* up to 14 bytes of direct address */
};
```

msghdr

This structure contains the message header for the **recvmsg** and **sendmsg** calls.

```
struct msghdr {
   caddr_t msg_name;              /* optional address            */
   int     msg_namelen;           /* size of address             */
   struct  iovec *msg_iov;        /* scatter/gather array        */
   int     msg_iovlen;            /* # elements in msg_iov       */
   caddr_t msg_accrights;         /* ignored for AF_INET         */
   int     msg_accrightslen;      /* ignored for AF_INET         */
};
```

clientid

This structure is used by the **getclientid**, **givesocket**, and **takesocket** routines. These routines allow MVS or CMS programs to pass socket descriptors. The routines compensate for the unavailablity of **fork** (except via OpenEdition) as a means of creating an independent process that shares file descriptors with its parent.

```
struct clientid {
   int domain;          /* Such as AF_INET.                       */
   char name[8];        /* Address space or virtual machine name. */
   char subtaskname[8]; /* Can be set to blank or a subask ID.    */
   char reserved20[20]; /* Do not use the contents of this        */
                        /* structure.                             */
};
```

<netdb.h>

This header file contains structures returned by the network database library. Internet addresses and port numbers are stored in network byte order, identical to IBM 370 byte order. Other quantities, including network numbers, are stored in host byte order. Despite the fact that network byte order and host byte order are identical on the IBM System/370, a portable program must distinguish between the two. The structures in the **<netdb.h>** header file are described in the following sections.

hostent

This structure contains host information.

```
struct hostent {
   char    *h_name;        /* official name of host               */
   char    **h_aliases;    /* alias list                          */
   int     h_addrtype;     /* host address type                   */
   int     h_length;       /* length of address                   */
   char    **h_addr_list;  /* list of addresses from name server  */
 #define h_addr h_addr_list[0]  /* address, for backward          */
                                /* compatiblity                   */
};
```

netent

This structure contains network information.

```
struct  netent {
   char            *n_name;        /* official name of network     */
   char            **n_aliases;    /* alias list                   */
   int             n_addrtype;     /* net address type             */
                                   /* Only AF_INET is supported.   */
   unsigned long   n_net;          /* network number               */
};
```

servent

This structure contains service information.

```
struct  servent {
   char    *s_name;        /* official service name                */
   char    **s_aliases;    /* alias list                           */
   int     s_port;         /* port #                               */
   char    *s_proto;       /* protocol to use                      */
};
```

protoent

This structure contains protocol information.

```
struct  protoent {
   char    *p_name;        /* official protocol name               */
   char    **p_aliases;    /* alias list                           */
   int     p_proto;        /* protocol #                           */
};
```

rpcent

<netdb.h> also defines a structure used to store information about programs using the Sun RPC protocol. This structure is defined as follows:

```
struct rpcent {
   char    *r_name;        /* name of server for RPC program       */
   char    **r_aliases;    /* alias list                           */
   int     r_number;       /* RPC program number                   */
};
```

Sun RPC programs can be identified by name or by number. The **getrpcbyname** function returns **rpcent** structure.

herror

The **<netdb.h>** header file also contains macro definitions for the integer **h_errno**, which describes name server conditions. Refer to Chapter 18, “Socket Function Reference” on page 18-1 for more information on **h_errno**.

<netinet/in.h>

This header file contains constants and structures defined by the Internet system. Several macros are defined for manipulating Internet addresses. Among these are **INADDR_ANY**, which indicates that no specific local address is required, and **INADDR_NONE**, which generally indicates an error in address manipulation functions. Refer to “bind” on page 18-6 for more information on binding a local address to the socket.

You must include the **<sys/types.h>** header file before this header file. The structures in the **<netinet/in.h>** header file are described in the following sections.

in_addr

This structure contains the Internet address in network byte order, which is the same as host byte order on the IBM System/370.

```
struct in_addr {
   u_long s_addr;
};
```

sockaddr_in

This structure contains the socket address, which includes the host's Internet address and a port number. This is the specific address structure used for a socket address when the transport mechanism is TCP/IP.

```
struct sockaddr_in {
   short   sin_family;
   u_short sin_port;
   struct  in_addr sin_addr;
   char    sin_zero[8];
};
```

<netinet/in_systm.h>

This header file contains definitions to facilitate the porting of low-level network control and query Internetwork Control and Message Protocol (ICMP), and Internetwork Protocol (IP) raw socket type applications. The Internet ping client utility is an example of such a program.

<netinet/in_systm.h> must usually be included before other ICMP or IP socket related header files such as **<netinet/ip.h>** and **<netinet/ip_icmp.h>**. Refer to the individual header file descriptions that follow for the specific dependency.

This header declares the following **typedef** names for use as abbreviations for three commonly used types for internetwork order, that is types with the bytes in high endian order.

typedef u_short n_short;
declares unsigned short integer as received from the network.

typedef u_long n_long;
declares an unsigned long integer as received from the network.

typedef u_long n_time;
declares an unsigned long integer representing minutes and seconds since 00:00 Greenwich mean time, in byte reverse order.

The following definition is also available to kernel functions that provide a network time definition for the **iptime** function.

```
#ifdef _KERNEL
n_time iptime();
#endif
```

<netinet/ip_icmp.h>

This header file contains definitions of constants and structures required for using the ICMP protocol as described in IBM's RFC 792. Prior inclusion of **<netinet/in_systm.h>** is required.

<netinet/ip.h> This header file contains definitions of constants and structures required for using the IP protocol (Internet Protocol, Version 4) as described in IBM's RFC 791. Prior inclusion of **<netinet/in_systm.h>** is required.

<netinet/udp.h> This header file contains definitions of the User Datagram Protocol (UDP) header for UDP datagrams. UDP datagrams consist of a fixed header section immediately followed by the data section. The length of entire datagram, including the header and data, is maintained in UDP length field as a count of the number of octets (an *octet* is 8 bits; on IBM S370/390 systems this is 1 byte) in the datagram. Thus, the mininum value is **8** (64 bits), which is the length of the header alone.

```
struct udphdr {
    u_short uh_sport;  /* source port      */
    u_short uh_dport;  /* destination port */
    short uh_ulen;     /* udp length       */
    u_short uh_sum;    /* udp checksum     */
};
```

<arpa/inet.h> This header file contains declarations for the network address resolution functions. You must include the **<netinet/in.h>** header file before this header file.

<arpa/nameser.h> This header file contains definitions that enable applications to communicate with Internet name servers. The contents of this header file are not of interest to most applications; however, this header file must be included before the **<resolv.h>** header file. Applications that manipulate resolver options must include this header file. You must include the **<sys/types.h>** header file before this header file.

<resolv.h> This header file contains global definitions for the resolver. Definitions and structures in the **<resolv.h>** header file are discussed in the following sections. You must include the **<sys/types.h>**, **<netinet/in.h>**, and **<arpa/nameser.h>** header files before this header file.

state

_res refers to a **state** structure describing resolver operations. In the SAS/C implementation, **_res** is a macro. Because **_res** is not a variable, a program should not directly declare it. Inclusion of the **<resolv.h>** header file declares **_res**.

```
struct state {
   int     retrans;               /* retransmition time interval     */
   int     retry;                 /* number of times to              */
                                  /* retransmit                      */
   long    options;               /* option flags - See below.       */
   int     nscount;               /* number of name servers          */
   struct  sockaddr_in nsaddr_list[MAXNS];  /* address of name       */
                                            /* server                */
 #define nsaddr  nsaddr_list[0]   /* for backward compatibility      */
   u_short id;                    /* current packet id               */
   char    defdname[MAXDNAME];    /* default domain                  */
   char    *dnsrch[MAXDNSRCH+1];  /* components of domain to search  */
};
```

Bitwise OR Options

The following bit masks are resolver options that can be specified. They are stored in `_res.options`.

`RES_DEBUG`
print resolver debugging messages.

`RES_USEVC`
use TCP connections rather than UDP datagrams for queries.

`RES_STAYOPEN`
when specified along with `RES_USEVC`, keep the TCP connection open between queries.

`RES_RECURSE`
for queries, set the recursion-desired bit. This is the default. `res_send` does not make queries iteratively. The name server handles recursion.

`RES_DEFNAMES`
have `res_mkquery` append the default domain name to single-component names. This is the default.

`RES_DNSRCH`
have `gethostbyname` search for host names in the current parent domains.

`RES_IGNTC`
ignore truncation errors; do not retry.

<net/if.h> This header file contains structures that define the network interface and provide a packet transport mechanism. `<net/if.h>` is useful only for low-level programming of the network interface. You must include the `<sys/types.h>` and `<sys/socket.h>` header files before this header file.

<strings.h> This header file provides compatibility with the BSD UNIX `<strings.h>` header file and the `index`, `rindex`, `bzero`, `ffs`, and `bcmp` functions. Refer to "BSD Library Dependencies" on page 16-4 for more information on using these functions.

Socket Functions

The following scenario depicts a typical sequence of events in a network application using stream sockets:

1. A server application issues a `socket` call, which returns an integer that is the socket descriptor, allocated to the AF_INET addressing family.
2. The server uses the `bind` function to bind a name to the socket and allow the network access to the socket.
3. The server issues the `listen` call to signal the network that it is ready to accept connections from other applications.
4. The server issues an `accept` call to accept the connection request.
5. A client application issues a `connect` call to the server.
6. The client and server conduct read/write operations.
7. The socket is disconnected with a `close` call.

Datagram sockets do not have to be bound or connected, but they normally need to acquire additional address information in order to communicate with each other.

The following sections describe the socket functions that gather information, convert data, and accompany each step in the process of communicating between applications over a network. Refer to Chapter 18, "Socket Function Reference" on page 18-1 for a complete description of each socket function.

Addresses and Network Information

The complexity of network communications frequently requires that an application have some means of determining the location of sockets or other applications and of acquiring information about the network.

Socket Address

A process may need to discover the address to which the socket is connected. The process may also need to discover its socket's local address. The following function calls determine socket addresses:

- **`getsockname`**
- **`getpeername`**.

Host Name

The following function calls return the host's assigned name and the host's Internet address:

- **`gethostname`**
- **`gethostid`**.

Address Manipulation

The following socket functions perform a translation between 32-bit IP addresses and the standard network dotted decimal notation, or they divide or reassemble the network and host sections of 32-bit IP addresses:

- **`inet_addr`**
- **`inet_lnaof`**
- **`inet_makeaddr`**
- **`inet_netof`**
- **`inet_network`**
- **`inet_ntoa`**.

Host Information

An application can obtain information about a host by submitting either the host's name or its address. The following functions return the host name and IP address:

- **`gethostbyname`**
- **`gethostbyaddr`**.

Network Information

An application can also obtain information about a network. The following function calls return network names and addresses:

- **`getnetbyname`**
- **`getnetbyaddr`**.

Protocol Information

An application can also obtain information about protocols. The following function calls return protocol names and numbers:

- **`getprotobyname`**
- **`getprotobynumber`**.

Network Services

An application can obtain information about network services and their protocol ports. The following functions return network service names and ports:

- **getservbyname**
- **getservbyport**.

Database Services

There are three sets of library routines that access a database to return information such as the names of machines and network services, and protocol port numbers. Generally speaking, these routines are of use only to applications, such as network management programs, that are intended to read the entire database.

With each set, an application can connect to a database, return a database entry, and disconnect. The pattern for the names of these routines is as follows:

- **setXent**
- **getXent**
- **endXent**.

X is the name of the database.

The following function calls perform database services:

- **sethostent**
- **gethostent**
- **endhostent**
- **setservent**
- **getservent**
- **endservent**
- **setprotoent**
- **getprotoent**
- **endprotoent**
- **setnetent**
- **getnetent**
- **endnetent**.

Resolver Routines

These routines make, send, and interpret data packets for use with the Internet Domain Name Service. The resolver consists of the following socket functions:

- **dn_comp**
- **dn_expand**
- **res_init**
- **res_mkquery**
- **res_send**.

The following macros and functions place long and short integers in a buffer in the format expected by the Internet name server:

- **GETSHORT/_getshort**
- **GETLONG/_getlong**
- **PUTSHORT/putshort**
- **PUTLONG/putlong**.

Data Conversion

Address or network information may need to be translated, and character sets received through a socket may need to be converted.

Byte Order Conversion

When an integer is copied from a network packet (unit of data) to a local machine or from a local machine to a network packet, the byte order must be converted between local machine byte order and network standard byte order. The following socket functions take a value as an argument and return that value with the bytes rearranged:

- **htons**
- **htonl**
- **ntohl**
- **ntohs**.

ASCII-EBCDIC Translation

The following functions translate between the ASCII character set generally used for text on the Internet, and the MVS or CMS EBCDIC character set:

- **ntohcs**
- **htoncs**.

Choosing a Socket Implementation

The SAS/C Socket Library supports both integrated and non-integrated sockets. Integrated sockets are available with OpenEdition and provide a higher degree of UNIX compatiblity. Non-integrated sockets rely on a run-time library implemented interface to TCP/IP software. The socket implementation is specified by the following function:

- **setsockimp**.

Creating a Socket

In order to create a socket, the application issues a **socket** call, specifying the domain and family to which the socket belongs. The program that creates the socket can also control factors such as length of timeout, type of data transmitted, and size of the communication buffer. Finally, the program can set operating characteristics of the socket such as whether an application can continue processing without becoming blocked during a **recv** call.

The following socket functions create a socket or control its operating characteristics:

- **socket**
- **socketpair**.
- **ioctl**
- **fcntl**
- **getsockopt**
- **setsockopt**.

Binding a Socket

A socket is created without associating it to any address. A local address can be bound to the socket with the following socket function:

- **bind**.

Listening for a Connection

When an application in the role of server has opened a stream socket, it binds the socket to a local address and then waits for a connection request from another application. To avoid confusion in message delivery, the application can set up a queue for the incoming connection requests. The following socket function prepares the server to await incoming connections:

- **listen**.

Connecting and Passing a Socket

Connecting a socket binds it to a permanent destination. If an application seeks to transmit data to a stream socket, it must first establish a connection to that socket. A server program can also pass sockets between two client programs. The following socket functions establish or pass connections between applications:

- `connect`
- `givesocket`
- `takesocket`
- `getclientid`.

Accepting a Connection

After a server has established a socket, it waits for a connection. The following socket function causes the application to wait for a connect request:

- `accept`.

Sending and Receiving Data Through a Socket

After a socket has been established, an application can transmit or receive data through the socket. The following socket functions are used in the transmission or receipt of data between sockets:

- `read`
- `readv`
- `recv`
- `recvfrom`
- `recvmsg`
- `selectecb`
- `send`
- `sendto`
- `sendmsg`
- `write`
- `writev`
- `select`.

Closing the Connection

An application closes a connection when it has no more data to send. An application can also shut down transmissions in one or both directions. The following functions close the connection or shut down transmissions:

- `close`
- `shutdown`
- `socktrm`.

16 Porting UNIX Socket Applications to the SAS/C® Environment

16-1 Introduction
16-1 Integrated and Non-Integrated Sockets
16-2 Socket Library Restrictions
16-2 Socket Descriptors
16-2 Addressing Families
16-2 Sockets
16-2 Function Names
16-3 Header Filenames
16-4 errno
16-4 BSD Library Dependencies
16-5 INETD and BSD Kernel Routine Dependencies
16-6 Character Sets
16-6 The Resolver

Introduction

The SAS/C Socket Library provides the highest practical level of compatibility with the BSD UNIX socket library for MVS and CMS environments. Programs whose only UNIX dependencies are in the area of sockets or other UNIX features already supported by the SAS/C Library can be compiled and run with little or no modification.

Because the socket functions are integrated with the existing SAS/C Library and are not add-on features, many of the incompatibilities of other MVS and CMS socket implementations have been avoided. For example, there are no requirements for additional header files that are specific to MVS or CMS environments; **`errno`**, and not some other variable specific to MVS or CMS, is used for socket function error codes, and the **`close`**, **`read`**, and **`write`** calls operate on both files and sockets, just as they do in a UNIX operating system.

There are still some areas where compatibility between MVS and CMS and the UNIX operating system is not possible. This chapter describes the areas of incompatibility that may cause problems when porting socket code from UNIX socket implementations to the SAS/C environment.

Integrated and Non-Integrated Sockets

Under the MVS/ESA 5.1 operating system, OpenEdition supports integrated sockets. This feature provides a TCP/IP socket interface that is integrated with OpenEdition support instead of being an interface to TCP/IP software implemented only in the run-time library. When you use integrated sockets, an open socket has an OpenEdition file descriptor, which can be used like any other OpenEdition file descriptor. For instance, unlike a non-integrated socket, an integrated socket remains open in a child process created by **`fork`**, or in a program invoked by an **`exec`** function. Thus, when integrated sockets are used, a higher degree of UNIX compatibility is available than when non-integrated sockets are used.

You must decide whether your application is going to use integrated or non-integrated sockets. For example, an application that may run on a system that does not support OpenEdition should use non-integrated sockets. The **`setsockimp`**

function specifies whether integrated or non-integrated sockets are being used. This function must be called before any other socket-related functions are called. By default, integrated sockets are used with **exec**-linkage applications, and non-integrated sockets are used otherwise.

Socket Library Restrictions

While almost every socket-related BSD function is available in the SAS/C Library, not all of the traditional UNIX features of these functions are available. The descriptions in Chapter 18, "Socket Function Reference" on page 18-1, describe the features of each function. This section contains a summary of the most significant restrictions.

Socket Descriptors

With Release 6.00 of the SAS/C Compiler, socket descriptors are assigned from the same range and according to the same rules as UNIX file descriptors. This aids in the porting of socket applications, since many such applications depend on this particular assignment of socket numbers. If OpenEdition is installed and running, the maximum number of open sockets and hierarchical file system (HFS) files is set by the site; the default is 64. If OpenEdition is not installed or not active, the maximum number of open sockets is 256. Note that programs written for previous releases of SAS/C software, which assume that socket numbers range from 256 to 511, may need to be modified to accommodate UNIX compatible socket number assignment.

Addressing Families

The BSD socket library design supports the use of more than one type of transport mechanism, known as the addressing family. UNIX implementations usually support at least two addressing families: **AF_INET** and **AF_UNIX**. **AF_INET** uses TCP/IP to transport data. **AF_UNIX** transports the data using the UNIX file system.

With integrated sockets, either **AF_INET** or **AF_UNIX** can be used. With non-integrated sockets, only **AF_INET** can be used. Programs that use **AF_UNIX** can usually be modified to use **AF_INET**.

Sockets

Many of the restrictions in the use of UNIX features are caused by the underlying TCP/IP implementation. These restrictions may vary, depending on the TCP/IP vendor and release. Vendor-specific restrictions affect the following:

- socket types. Types other than SOCK_STREAM and SOCKET_DGRAM may not be supported.
- socket options used by the **setsockopt** and **getsockopt** functions.
- **fcntl** commands.
- **ioctl** commands.
- **errno** values.

In addition, there are the following general restrictions:

- Asynchronous I/O to sockets is not supported. (Non-blocking I/O is supported.)
- The **socketpair** function is supported only with integrated sockets.

Function Names

UNIX operating systems support very long external names. The MVS and CMS linking and library utilities restrict external names to eight characters. The SAS/C Compiler and COOL utility can map long external names into eight-character names, making the external MVS and CMS restrictions invisible in most cases.

The SAS/C Library also supports the **#pragma map** statement, which directs the compiler to change an external name in the source to a different name in the object file. Thus, long names are shortened, and names that are not lexically valid in C language can be generated. The socket library header files change long socket function names by including **#pragma map**. For example, the **<netdb.h>** header file contains the following statement:

```
#pragma map (gethostbyname, "#GHBNM")
```

This statement changes the **gethostbyname** function to **#GHBNM**. The **#pragma map** statement enables you to use TCP/IP functions with long names in your source and does not require that you use the extended names option or the COOL utility.

#pragma map statements are already in the header files required for each function. You normally do not have to modify your source to accommodate long names.

If your program produces an unresolved external reference for a socket function containing a long name, first make sure that you have included the appropriate header files as listed in the description for each socket function in Chapter 18, "Socket Function Reference" on page 18-1. The following header files are not always required by UNIX C compilers but are required by the SAS/C Compiler to resolve long names:

- Include the **<arpa/inet.h>** header file with the **inet_*** functions, such as **inet_addr**. This is correct coding practice but is not required by UNIX C compilers. As a compatibility feature, the SAS/C Library file **<netinet/in.h>** includes the **<arpa/inet.h>** file.
- Include the **<netdb.h>** header file with the **gethostid** and **gethostname** functions. This is not required by UNIX C compilers. To reduce incompatibilities caused by failure to include the **<netdb.h>** header file in existing source code, **#pragma map** statements for these functions are also available in the **<sys/types.h>** header file.
- The **<socket.h>**, **<netdb.h>**, **<resolv.h>**, and **<nameser.h>** header files all contain **#pragma map** statements for the functions that require them. Most UNIX programs include these headers with the appropriate functions.

The functions in most programs ported from a UNIX operating system have long names. If a function in your program contains a long name, use the **extname** name compiler option and the COOL utility to compile and link your program. The effects of both the **#pragma map** statement and the **extname** option are not usually visible to the user. For information on the significance of these features during machine-level debugging, reading link-edit maps, and writing zaps, refer to Appendix 7, "Extended Names," in the *SAS/C Compiler and Library User's Guide.*

Header Filenames

UNIX header filenames are really pathnames that relate to the **/usr/include** directory. In most cases, the headers reside directly in the **/usr/include** directory with no further subdirectories in the pathname. For example, the **<netdb.h>** header file resides in the **/usr/include** directory. MVS and CMS file structures do not include subdirectories. All angle-bracketed include files are in the SYSLIB concatenation under MVS or in the GLOBAL MACLIB concatenation under CMS. The SAS/C Compiler ignores subdirectories included in the filename. Specifying **<sys/socket.h>** appears the same to MVS and CMS systems as specifying **<socket.h>**.

Header files such as **<arpa/nameser.h>** and **<sys/socket.h>** are placed in an MVS partitioned data set or CMS macro library based on the last part of the filename, for example, **socket.h**. Because of this, a UNIX program that specifies a subdirectory in the header file pathname can work without modification. It is best to code the pathname even for programs intended for use with SAS/C software because they can be ported back to a UNIX operating system more easily and because future releases of the SAS/C Compiler may attach significance to these pathnames. The header files listed with the socket function descriptions in Chapter 18, "Socket Function Reference" on page 18-1 include subdirectories.

errno

When there is an error condition, most socket functions return a value of -1 and set **errno** to a symbolic value describing the nature of the error. The **perror** function prints a message that explains the error. The SAS/C Library adheres as closely as possible to symbolic UNIX **errno** values. However, except for specifically defined **errno** values, such as **EWOULDBLOCK**, programs may not receive exactly the same **errno** values as programs would in a particular implementation of the UNIX operating system. The message printed by the **perror** function may also differ.

Because **errno** is a macro and not a simple external variable, you should always declare it by including the **<errno.h>** header file.

Two external symbols, **h_errno** and **_res**, are defined parts of the network database and resolver interfaces. **h_errno** values are the same as those in common versions of the UNIX operating system, but the **herror** text may be different. As with **errno**, you cannot declare these symbols directly. Always declare them by including the appropriate header file.

BSD Library Dependencies

Many socket programs implicitly assume the presence of the BSD library. For example, the BSD function **bcopy** is widely used in socket programs even though it is not portable to UNIX System V. Because of the lack of acceptance of such routines outside of the BSD environment and the fact that the same functionality is often available using ANSI Standard routines, BSD string and utility functions have not been added to the SAS/C Library.

Many UNIX programs ported to the SAS/C Library already contain support for System V environments in which Berkeley string and utility routines are not available. These programs usually call ANSI Standard functions instead. Using ANSI Standard functions is the best means of porting UNIX programs to the SAS/C Library because common ANSI string functions are often built in, and because the code will be more portable to other environments.

To ease porting of code that relies on Berkeley string and utility routines, the SAS/C Usage Notes sample library contains the member BSDSTR, which includes sample source code for the following functions:

- **bcopy**
- **bzero**
- **bcmp**
- **index**
- **rindex**
- **ffs**.

The BSD **<strings.h>** header file is also available to facilitate the compilation of programs that rely on Berkeley string and utility routines. By default, the **<strings.h>** header file does not define macros for the functions in the previous list because problems arise in compiling programs that contain direct declarations of the functions. If your program does not contain direct declarations of functions, you can use the **#define** option to define **_BSD_MACROS** before you include the **<strings.h>** header file. Refer to Chapter 7, "Compiler Options," in the *SAS/C Compiler and Library User's Guide* for information on the **define** compiler option.

INETD and BSD Kernel Routine Dependencies

One of the greatest hurdles to overcome in porting some BSD socket programs is their dependence on BSD kernel routines, such as **fork**, that are not supported by the SAS/C Compiler (except under OpenEdition MVS). This level of dependency is greatest in BSD daemon programs called from INETD, the UNIX TCP/IP daemon.

Except under OpenEdition, SAS/C software does not support the following UNIX kernel routines that are commonly used in TCP/IP daemon processes and other UNIX programs:

fork In the UNIX operating system, the **fork** system call creates another process with an identical address space. This process enables sockets to be shared between parent and child processes. The **fork** function is available under OpenEdition, and UNIX socket behavior occurs if integrated sockets are specified. However, creating an identical address space this way is not possible under traditional MVS or CMS, although the **ATTACH** macro may be used under MVS to achieve similar results.

exec Under UNIX, the **exec** system call loads a program from an ordinary, executable file onto the current process, replacing the current program. With OpenEdition, the **exec** family of functions may be used to create a process, and the UNIX socket behavior occurs if integrated sockets are specified. Under traditional MVS or CMS, the ISO/ANSI C **system** function sometimes can be used as an alternative to the **exec** routine, but the semantics and effects are different.

dup,**dup2** Unlike UNIX style I/O, standard I/O is the most efficient and lowest level form of I/O under MVS and CMS because of the implementation-defined semantics of ISO/ANSI C. The looser semantics place standard I/O closer to native MVS and CMS I/O than to UNIX style I/O. The inverted relationship between UNIX style I/O and standard I/O inhibits **dup** implementation under traditional MVS and CMS. However, with OpenEdition, **dup** and

	dup2 are available, and UNIX socket behavior occurs if integrated sockets are specified.
socketpair, **pipe**	The socket pair and pipe calls are not useful without the **fork** system call.

Daemons created by INETD depend heavily on the UNIX environment. For example, they assume that a **dup** call has been issued to correspond to the **stdin**, **stdout**, and **stderr** file pointers. This correspondence relies on the way the **fork** system call handles file descriptors.

System programs that involve the INETD daemon must be redesigned for MVS or CMS. The SAS/C Library provides the **givesocket**, **takesocket**, and **getclientid** functions to allow a socket to be passed between cooperating processes in the absence of the **fork** system call. Refer to Chapter 18, "Socket Function Reference" on page 18-1 for more information on these functions.

Character Sets

EBCDIC-to-ASCII translation is one of the greatest sources of socket program incompatibility. When communicating with a program in almost any environment other than MVS and CMS, text must be translated from EBCDIC into ASCII. If all transmitted data were text, the SAS/C Library could translate text automatically to ASCII before sending the data, and it could translate the text to EBCDIC automatically when receiving data. Unfortunately, only the program knows which data are text and which data are binary. Therefore, the program must be responsible for the translation.

The SAS/C Library provides the **htoncs** and **ntohcs** routines to facilitate EBCDIC-to-ASCII translation. ASCII is the character set used in network text transmission. The **htoncs** and **ntohcs** routines are not portable to UNIX, but you can define them as null function-like macros for environments other than CMS and MVS. You can recompile these routines if you want to use a different EBCDIC-to-ASCII translation method.

Note that, except when using the resolver (see the following section, "The Resolver," for more information), the SAS/C Library does not perform any translations from ASCII to EBCDIC.

The Resolver

In addition to the standard communication and network database routines in the UNIX environment, the SAS/C Library contains a complete implementation of the BSD resolver and provides the standard UNIX interface for resolver programming. This facilitates the writing of applications that communicate with Internet name servers. The resolver is compatible with the UNIX operating system because the routines are derived from the BSD network source. There are, however, three compatibility issues that should be considered:

- ASCII-to-EBCDIC translation is performed automatically by the **dn_expand** function, and EBCDIC-to-ASCII translation is performed by the **dn_comp** function. These translations should resolve any ASCII-to-EBCDIC translation problems for domain names without requiring special code in the application. The SAS/C Library can translate automatically in this instance because it recognizes that the data are intended to be ASCII text.

- Routines that are declared to be external in the BSD name server but that are not documented in the UNIX man pages (for example, the routines that print resolver debugging information) cannot be called directly in the SAS/C implementation.
- The **`_res`** variable cannot be declared directly in a program because it is implemented as a macro in SAS/C software. Include the **`<resolv.h>`** header file for a definition for the **`_res`** variable.

17 Network Administration

17-1 Introduction
17-1 Configuration Data Sets
17-1 Search Logic
17-2 Finding the Data Set
17-3 Specifying TCPIP_PREFIX for MVS
17-4 Specifying TCPIP_MACH
17-4 gethostbyname and Resolver Configuration
17-5 Configuring for the CMS Environment
17-5 Configuring for the MVS Environment

Introduction

The operation of the SAS/C Socket Library depends on its ability to access the configuration information for a site. In some cases, the socket library locates the information automatically. In other cases, you may need to specify the location of the information.

This chapter discusses the location of site configuration files and provides a detailed explanation of how your SAS/C Socket Library finds these files.

Configuration Data Sets

Under UNIX operating systems, these six data sets usually contain site-dependent configuration information for TCP/IP:

- **`/etc/hosts`**
- **`/etc/networks`**
- **`/etc/services`**
- **`/etc/protocols`**
- **`/etc/resolv.conf`**
- **`/etc/rpc`**.

The socket library uses equivalent data sets under the MVS or CMS operating systems. MVS and CMS file systems differ from the UNIX file structure, and local security or organization considerations can affect how data sets are named. Although the name of the data set may be different, the data set that contains site configuration information is in the same format as the equivalent data set under the UNIX operating system. If data sets in the UNIX format are not available, the library sometimes attempts to determine site information from vendor-specific data sets.

Search Logic

Under the MVS and CMS operating systems, the data set that contains configuration information usually has a name that is derived from the equivalent UNIX filename. For example, the MVS data set name ETC.HOSTS is derived from the UNIX filename **`/etc/hosts`**.

The socket library uses the following search logic when searching for the data set containing configuration information:

1. It determines the name by using an environment variable if one has been set. The name should begin with a SAS/C prefix, such as **`DSN:`** for a data set name,

or **HFS:** for a hierarchical file system (HFS) name. If the name does not begin with a prefix, it is interpreted according to the SAS/C Library defaults, which vary from program to program. For example, the default prefix for MVS is DDN:, and for CMS it is CMS:. Use the DSN: prefix if you are supplying the name in DSName form.

2. Under TSO, it searches for a data set name that is composed of the user's userid (TSO: prefix) and the name derived from the UNIX filename.
3. It searches for a data set name derived solely from a UNIX filename. Under MVS, the library does not perform this step if the **TCPIP_PREFIX** environment variable is not TCPIP. See "Specifying TCPIP_PREFIX for MVS" on page 17-3 for a discussion of **TCPIP_PREFIX**.
4. Under MVS, it searches for a data set derived from a UNIX filename and prefixed by the **TCPIP_PREFIX** value.

Finding the Data Set

The socket library uses the following methods to look for each of the configuration data sets:

/etc/protocols

The socket library looks for the following data set names while searching for the MVS or CMS data set that is equivalent to **/etc/protocols**:

1. value of **ETC_PROTOCOLS** environment variable, if defined
2. *tso-prefix*.ETC.PROTO under TSO
3. ETC.PROTO under MVS, or ETC PROTO under CMS
4. *tcpip-prefix*.ETC.PROTO under MVS, if **TCPIP_PREFIX** is not blank.

/etc/services

The socket library looks for the following data set names while searching for the MVS or CMS data set that is equivalent to **/etc/services**:

1. value of **ETC_SERVICES** environment variable, if defined
2. *tso-prefix*.ETC.SERVICES under TSO
3. ETC.SERVICES under MVS, or ETC SERVICES under CMS
4. *tcpip-prefix*.ETC.SERVICES under MVS, if **TCPIP_PREFIX** is not blank.

/etc/hosts

The socket library looks for the following data set names while searching for the MVS or CMS data set that is equivalent to **/etc/hosts**:

1. value of **ETC_HOSTS** environment variable, if defined
2. *tso-prefix*.ETC.HOSTS under TSO
3. ETC.HOSTS under MVS, or ETC HOSTS under CMS
4. *tcpip-prefix*.ETC.HOSTS under MVS, if **TCPIP_PREFIX** is not blank.

/etc/networks

The socket library looks for the following data set names while searching for the MVS or CMS data set that is equivalent to **/etc/networks**:

1. value of **ETC_NETWORKS** environment variable, if defined
2. *tso-prefix*.ETC.NETWORKS under TSO
3. ETC.NETWORKS under MVS, or ETC NETWORKS under CMS
4. *tcpip-prefix*.ETC.NETWORKS under MVS, if **TCPIP_PREFIX** is not blank.

/etc/resolv.conf

The socket library looks for the following data set names while searching for the MVS or CMS data set that is equivalent to `/etc/resolv.conf`:

1. value of **`ETC_RESOLV_CONF`** environment variable, if defined
2. *tso-prefix*.ETC.RESOLV.CONF under TSO
3. ETC.RESOLV.CONF under MVS, or ETC RESOLV under CMS
4. *tcpip-prefix*.ETC.RESOLV.CONF under MVS, if **`TCPIP_PREFIX`** is not blank.

/etc/rpc

The socket library looks for the following data set names while searching for the MVS or CMS data set that is equivalent to `/etc/rpc`:

1. value of **`ETC_RPC`** environment variable, if defined.
2. *tso-prefix*.ETC.RPC under TSO. Under MVS but not under TSO, it looks for *tcpip-prefix*.ETC.RPC if a userid can be determined for the address space.
3. ETC.RPC under MVS, or ETC RPC under CMS.
4. *tcpip-prefix*.ETC.RPC under MVS, if **`TCPIP_PREFIX`** is not blank.

When the socket library finds a data set with one of the above names, the name is retained for the duration of the program's execution. You may need to restart the program for the socket library to find a different filename.

Specifying TCPIP_PREFIX for MVS

The value of the **`TCPIP_PREFIX`** environment variable used in finding configuration data sets is determined in the following way:

1. If the value is set by the user, either interactively or through a program, before the socket library's initial attempt to use the **`TCPIP_PREFIX`** variable to locate a data set, the library uses the value set by the user.
2. If the **`TCPIP_PREFIX`** variable is undefined when the socket library attempts to use it to locate a data set, the **`TCPIP_PREFIX`** variable is set with a 27-character string array in L$CNDBA in the transient library. However, the library may search the TCPIP.DATA file for the DATASETPREFIX keyword value before using the array. (See item 2 under "gethostbyname and Resolver Configuration" on page 17-4 for search order.) The default value of this array is TCPIP. A zap that changes this default is provided with the transient library installation instructions. Some SAS/C programs received from software vendors or other sites may not use the transient library and could require their own zaps. For this reason, avoid using the zap, if possible, and instead use the ETC high-level qualifier derived from the UNIX filename or make TCPIP the default high-level qualifier.

For information on setting environment variables, see "Environment Variables" in Chapter 8, "Run-Time Argument Processing," in *SAS/C Compiler and Library User's Guide*.

Specifying TCPIP_MACH

For IBM TCP/IP, the socket library must locate a TCP/IP virtual machine under CMS or an address space under MVS. The name of the virtual machine or the address space may vary from site to site. The SAS/C Compiler uses the **TCPIP_MACH** environment variable to determine the value of this name. If the **TCPIP_MACH** variable does not exist, the socket library searches for the name in the TCPIP.DATA file under MVS or the TCPIP DATA file under CMS. (See item 2 under "gethostbyname and Resolver Configuration" on page 17-4 for search order.) If this file is not available, the socket library uses a default value of TCPIP to locate the TCP/IP virtual machine or address space.

gethostbyname and Resolver Configuration

Under the UNIX environment, the **gethostbyname** and **gethostbyaddr** routines may use the **/etc/hosts** file, or they may call the resolver to contact the name server for the host name information.

The SAS/C Socket Library uses the following logic when looking up host names and addresses:

1. looks for the **/etc/resolv.conf** file using the rules listed in "/etc/resolv.conf" on page 17-3. If the socket library finds the **/etc/resolv.conf** file, it performs the requested queries through the resolver, and it returns any answer it receives. If attempts to connect to name servers are refused (**errno ECONNREFUSED**), it goes to step 3.
2. looks for a data set in the format of the IBM TCP/IP file TCPIP.DATA under MVS, or TCPIP DATA under CMS. The search rules for this data set are those used by IBM TCP/IP. The socket library
    a. looks for the environment variable **TCPIP_DATA** string. If found, the string is passed to **fopen**
    b. looks for the data set identified by the DDname SYSTCPD. If found, the filename is passed to **fopen**
    c. looks for *tso-prefix*.TCPIP.DATA under TSO
    d. looks for the SYS1.TCPPARMS(TCPDATA) data set
    e. looks for the environment variable **TCPIP_PREFIX** and then searches for *tcpip_prefix*.TCPIP.DATA
    f. uses the default value of **TCPIP_PREFIX** and searches for *default-value*.TCPIP.DATA
    g. looks for TCPIP.DATA.
3. if attempts to connect to name servers defined in TCPIP.DATA are refused, the socket library looks for an **/etc/hosts** file using the rules listed in "/etc/hosts" on page 17-2. If the socket library finds an **/etc/hosts** file, it returns the result, including failure.

Determining the domain name in name-server queries follows the same logic as the UNIX operating system in using the **domain** statement of the **/etc/resolv.conf** file, the file specified by the **HOSTALIASES** environment variable and the value of the **LOCALDOMAIN** environment variable. Name-server addresses are also determined from the **/etc/resolv.conf** file.

If, because there is no **/etc/resolv.conf** file, an IBM TCP/IP TCPIP.DATA file is read, resolver configuration is determined by the statements, including IBM defaults, in the TCPIP.DATA file. The SAS/C Library only recognizes the first three name servers specified in this file. Both the UNIX operating system and the SAS/C environment have a limit of three name servers.

Configuring for the CMS Environment

Under the CMS environment, the easiest way to configure your system is to locate files that have names derived from UNIX filenames on a minidisk that is accessible to all TCP/IP users. If you want to use the resolver for name resolution, you should create an ETC RESOLV file. If you are running IBM TCP/IP, you can use the TCPIP DATA file instead.

Configuring for the MVS Environment

Under the MVS environment, the easiest way to configure your system is to give the configuration data sets the ETC high-level qualifier, for example, ETC.HOSTS and ETC.RESOLV.CONF. If you then do not set the **`TCPIP_PREFIX`** environment variable, and you do not apply the zap to the **`TCPIP_PREFIX`** in the transient library, your programs will always be able to find the configuration data sets. If you want to use the resolver for name resolution, you can create an ETC.RESOLV.CONF file.

An IBM TCP/IP site that prefers to use existing data sets can use the *tcpip-prefix*.TCPIP.DATA file to control name resolution.

An MVS site that does not use TCPIP as the high-level qualifier and that cannot use the ETC prefix will have to rely on environment variables (possibly **`DATASET_PREFIX`** in the TCPIP.DATA file) or the zap provided in the installation instructions. Environment variables work well if there is a way to set them, such as a CLIST that all TCP/IP users can run when they log on or before they run a client program. Also, using environment variables that have a permanent scope enables the user to set the variable once and then use the setting from that point onward. A site that cannot use the environment variables must rely on the zap provided in the installation instructions. Programs received from other sites may also require this zap.

18 Socket Function Reference

18-1 Introduction

Introduction

This chapter contains a description of each function in the SAS/C Socket Library, along with example code. Each description contains a definition of the function, a synopsis, discussions of return values and portability issues, and an example. Also, errors, cautions, diagnostics, implementation details, usage notes, and a list of related functions are included, if appropriate.

accept Accepts a Connection Request

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, void *addr, int *addrlen);
```

DESCRIPTION

accept extracts the first connection in the queue of connection requests, creates a new socket with the same properties as socket descriptor **s**, and allocates a new socket descriptor for the socket. Connection-based servers use **accept** to create a connection that is requested by a client. Socket **s** is a connection-based socket of type **SOCK_STREAM** that is bound to an address with a **bind** call and listens for connections with a **listen** call.

If there are no pending connection requests and **ioctl** or **fcntl** has not been used to designate the socket as non-blocking, **accept** blocks the caller until there is a connection. If the socket is designated as non-blocking, and there are no pending connection requests, **accept** returns a -1 and sets **errno** to **EWOULDBLOCK**; the new socket descriptor cannot be used to accept more connections. Socket descriptor **s** remains open.

addr and **addrlen** describe the buffer into which **accept** places the address of the new peer. **addr** can be **NULL**. If it is not **NULL**, it should point to a **sockaddr** structure or one of its derivatives, such as **sockaddr_in**.

addrlen points to an integer containing the size of the buffer in bytes. If the buffer size is not large enough to contain the address of the peer, the value of the address is not completely copied. No error occurs in this situation. On return, the integer pointed to by **addrlen** is set to the length that was actually copied.

If socket **s** is a member of the **read** descriptor set (**readfds**), you can use a **select** call to discover whether or not any connections are pending.

RETURN VALUE

If **accept** is successful, it returns a nonnegative value that is the descriptor of the newly created socket. Otherwise, it returns a **-1**.

PORTABILITY

accept is portable to other environments, including UNIX systems, that implement BSD sockets.

EXAMPLE

In the following example, **accept** is used to accept incoming connection requests.

```
/* This is a complete example of a simple server for the      */
/* daytime protocol. A daytime server waits on port           */
/* 13 and returns a human readable date and time string       */
/* whenever any client requests a connection.                 */
/* This server handles only TCP. The client does not need     */
/* to write to the server. The server responds with           */
/* the date and time as soon as the connection is made.       */
```

accept Accepts a Connection Request
(continued)

```
/* We will implement an iterative server (one which handles   */
/* connections one at a time) instead of a concurrent         */
/* server (one which handles several connections              */
/* simultaneously).                                           */
/* This program runs until it is stopped by an operating      */
/* system command.                                            */

#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <time.h>


   /* No need to translate characters on ASCII systems.       */
#ifndef  __SASC__
#define ntohcs(c) (c)
#define htoncs(c) (c)
#endif

   /* Define some useful constants.                           */
#define TRUE 1
#define DAYTIME_PORT 13

main()
{
      /* Socket descriptors. "s" will be the model descriptor */
      /* that indicates the type of connections that we       */
      /* want to accept. "cs" will be the socket used for     */
      /* communications with clients.                         */
   int s, cs;

      /* socket addresses for server and client               */
   struct sockaddr_in sa_serv;
   struct sockaddr_in sa_clnt;
   int sa_len;

      /* will contain information about the daytime service   */
   struct servent *serv;

      /* variables used to obtain the time                    */
   char *p;
   int len;
   time_t t;
```

accept Accepts a Connection Request
(continued)

```
   /* buffer for outgoing time string                      */
  char outbuf[128];

   /* Get a socket of the proper type. We use SOCK_STREAM  */
   /* because we want to communicate using TCP.            */
  s = socket(AF_INET, SOCK_STREAM, 0);
  if (s == -1) {
     perror("daytime - socket() failed");
     exit(EXIT_FAILURE);
  }

   /* Find the port for the daytime service. If the        */
   /* services file is not available, we will use the      */
   /* standard number. We specify "tcp" as the protocol.   */
   /* This call will attempt to find the services file.    */
  serv = getservbyname("daytime", "tcp");

   /* Prepare a socket address for the server. We specify  */
   /* INADDR_ANY instead of an IP address because we       */
   /* want to accept connections over any IP address       */
   /* by which this host is known. We specify the          */
   /* well-known port number for the daytime server if     */
   /* the program is compiled for a PRIVILEGED user        */
   /* (root on UNIX). Otherwise, we let TCP/IP select      */
   /* the port and then we print it so that clients will   */
   /* know what port to ask for.                           */
  memset(&sa_serv,'\0',sizeof(sa_serv));
  sa_serv.sin_family = AF_INET;
  sa_serv.sin_addr.s_addr = INADDR_ANY;
#ifdef PRIVILEGED
  sa_serv.sin_port = serv ? serv->s_port : htons(DAYTIME_PORT);
#else
  sa_serv.sin_port = 0;
#endif

   /* Bind our socket to the desired address. Now clients  */
   /* specifying this address will reach this server.      */
  if (bind(s, &sa_serv, sizeof(sa_serv)) == -1) {
     perror("daytime - bind() failed");
     return(EXIT_FAILURE);
  }

#ifndef PRIVILEGED
  sa_len = sizeof(sa_serv);
  if (getsockname(s, &sa_serv, &sa_len) == -1) {
     perror("daytime - getsockname() failed");
     return(EXIT_FAILURE);
  }
  printf("Daytime server port is: %d\n",
                 (int) ntohs(sa_serv.sin_port));
#endif
```

accept Accepts a Connection Request
(continued)

```
      /* Set up a queue for incoming connection requests.     */
   listen(s, SOMAXCONN);

      /* Accept incoming requests until cancelled by the      */
      /*   operating system or an error occurs.               */
   while (TRUE) {
         /* Accept a new request. Ask for client's address    */
         /* so that we can print it if there is an error.     */
      sa_len = sizeof(sa_clnt);
      cs = accept(s, &sa_clnt, &sa_len);
      if (cs==-1) {
        perror("daytime - accept() failed");
        return EXIT_FAILURE;
      }

         /* Send the time to the client. Daytime clients      */
         /* expect the string to be in ASCII.                 */
      time(&t);                    /* machine-readable time  */
      p = ctime(&t);               /* human-readable time    */
         /* Convert to ASCII if necessary. */
      for (len=0; p[len]  && len<sizeof(outbuf); len++)
         outbuf[len]  = htoncs(p[len] );

      if (write(cs,outbuf,len)==-1) {
         perror("daytime - write() failed");
         printf("Client IP address: %s\n",
                 inet_ntoa(sa_clnt.sin_addr));
         return EXIT_FAILURE;
      }

      close(cs);
   }
   return EXIT_SUCCESS;        /* Avoid compilation warnings. */
}
```

RELATED FUNCTIONS

bind, **connect**, **listen**, **select**, **socket**

bind Assigns a Name to a Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, const void *addr, int addrlen);
```

DESCRIPTION

bind assigns a name to an unnamed socket **s**. When a socket is created, it exists in an address family, but it does not have a name assigned to it. Servers use **bind** to associate themselves with a well-known port. Servers may also use **bind** to restrict access by other network addresses on a host with multiple network addresses. **bind** enables a connectionless client to have an address that the server can use for responses.

For addresses in the **AF_INET** family, **addr** points to a **sockaddr** or **sockaddr_in** structure. **addrlen** is the length of the address, in bytes. **addrlen** should be greater than or equal to the size of the **sockaddr** or **sockaddr_in** structure. The **INADDR_ANY** constant in the **<netinet/in.h>** header file specifies that the network address is not restricted. If the **sin_port** field of the **sockaddr** structure is zero, **bind** chooses a port. Alternatively, a well-known port number can be passed in the **sin_port** field. Internet host addresses and port numbers in **sockaddr_in** are always in network byte order. The remainder of the **sockaddr** or **sockaddr_in** structure should be **0**.

Upon return, if a port of **0** was specified, the selected port value is filled in. On return, the structure pointed to by **addr** should be the same as the structure pointed to by **getsockname** for this socket **s**.

RETURN VALUE

If **bind** is successful, it returns a **0**; otherwise, it returns a **-1** and sets **errno** to indicate the type of error.

PORTABILITY

bind is portable to other environments, including most UNIX systems, that implement BSD sockets.

EXAMPLE

In this example, **bind** assigns socket **s** to an arbitrary port without restriction by network address.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <string.h>
#include <stdio.h>
```

bind Assigns a Name to a Socket
(continued)

```
f()
{
   struct sockaddr_in sa;
   int s;
   struct servent *serv;
   .
   .
   .
      /* Specify the port. */
   memset(&sa,'\0',sizeof(sa));
   sa.sin_family = AF_INET;
   sa.sin_addr.s_addr = INADDR_ANY;
   sa.sin_port = serv->s_port;
   if (bind(s, &sa, sizeof(sa)) == -1) {
      perror("bind() failed");
      return -1;
   }
   .
   .
   .
      /* Let TCP/IP choose the port. */
   memset(&sa,'\0',sizeof(sa));
   sa.sin_family = AF_INET;
   sa.sin_addr.s_addr = INADDR_ANY;
   sa.sin_port = 0;
   if (bind(s, &sa, sizeof(sa)) == -1) {
      perror("bind() failed");
      return -1;
   }
   .
   .
   .
}
```

RELATED FUNCTIONS

connect, **getservbyname**, **getsockname**, **htons**

connect Associates a Socket with a Process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, const void *addr, int addrlen);
```

DESCRIPTION

connect attempts to associate socket **s** with a peer process at address **addr**. **addr** is a pointer to a buffer containing the address of the peer socket. **addrlen** is the length, in bytes, of the buffer pointed to by **addr**. **addrlen** should be greater than or equal to the number of bytes in the **sockaddr** or **sockaddr_in** structure.

For connection-oriented protocols on blocking sockets, when the establishment of a connection is actually attempted, the call blocks I/O until the connection attempt succeeds or fails. On non-blocking sockets, the call returns immediately, with **errno** set to **EINPROGRESS** if the connection could not complete immediately. The caller can then discover whether or not the connection is complete by issuing the **select** call to determine if the socket is ready for writing.

For sockets that use connectionless protocols, **connect** enables the socket to register a destination address once, instead of having to specify the same destination address for every read or write operation. By associating a packet with a specific socket, **connect** provides more information with which to trace the source of the problem if a transmission fails. In this case, **connect** can be called many times.

RETURN VALUE

If **connect** is successful, it returns a **0**; otherwise, it returns a **-1** and sets **errno** to indicate the type of error.

If **errno** is set to **ECONNREFUSED**, reuse of the failed socket descriptor is TCP/IP implementation-specific and unpredictable. However, it is always safe to close and obtain another socket descriptor for subsequent calls to **connect**.

PORTABILITY

connect is portable to other environments, including most UNIX systems, that implement BSD sockets.

EXAMPLE

In this example, **connect** connects socket **s** to a specific host and port.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
f()
```

connect Associates a Socket with a Process
(continued)

```
{
   struct sockaddr_in sa;
   struct hostent *host;
   struct servent *serv;
   int s;
   .
   .
   .
      /* Specify destination socket address (IP address and */
      /* port number).                                      */
   memset(&sa,'\0',sizeof(sa));
   sa.sin_family = AF_INET;
   memcpy(&sa.sin_addr,host->h_addr,sizeof(sa.sin_addr));
   sa.sin_port = serv->s_port;
      /* Connect to the host and port.                      */
   if (connect(s, &sa, sizeof(sa)) == -1) {
      perror("connect() failed");
      return -1;
   }
   .
   .
   .
}
```

RELATED FUNCTIONS

accept, **select**, **socket**, **getpeername**

dn_comp Translates Domain Names to Compressed Format

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_comp(char *exp_dn, char *comp_dn, int length, char **dnptrs,
            char **lastdnptr);
```

DESCRIPTION

dn_comp is part of the resolver, which is a set of routines that provides a programming interface for communicating with Internet name servers. **dn_comp** translates domain names in conventional character string format to the compressed format used by name servers. The compression process merges common suffixes among the names included in a name server query.

exp_dn is an EBCDIC string that contains a DNS name, such as a host name. **dn_comp** stores the equivalent compressed string in the buffer pointed to by **comp_dn**. **length** is the size of this buffer in bytes. **dn_comp** also maintains a list of compressed name elements. This list is used as a reference to eliminate common suffixes during a series of calls to **dn_comp** when multiple names are stored in the same buffer.

dnptrs points to the beginning of an array of pointers that points to the list of compressed name elements. The calling program allocates this array by using a convenient size, such as 10 elements.

lastdnptr points to the last element of the array. **dnptrs[0]** should point to the beginning of the message. Initially, **dnptrs[1]** should be **NULL**.

In the interests of greater portability, the SAS/C version of **dn_comp** performs EBCDIC-to-ASCII translation of **exp_dn** before beginning its compression process. If **dnptr** is **NULL**, the domain name is not compressed. Alternatively, if **lastdnptr** is **NULL**, the list of labels is not updated.

For information on the UNIX programming interface and Internet name servers, refer to "The Domain Name System" and "The Socket Interface" in *Internetworking with TCP/IP, Volume I*.

RETURN VALUE

If **dn_comp** is successful, it returns the size of the compressed domain name. Otherwise, it returns a **-1** and sets **errno** to indicate the type of error.

PORTABILITY

dn_comp is available on most versions of the UNIX operating system.

IMPLEMENTATION

The SAS/C version of **dn_comp** is a direct port from the BSD UNIX socket library. The EBCDIC-to-ASCII translation feature is the only change.

RELATED FUNCTIONS

dn_expand, **res_mkquery**

dn_expand Expands Compressed Domain Names

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int dn_expand(char *msg, char *eomorig, char *comp_dn, char *exp_dn,
              int length);
```

DESCRIPTION

dn_expand expands the compressed domain name to a full domain name. Expanded names are converted to uppercase EBCDIC.

msg
is a pointer to the beginning of the domain name message that contains the name to be expanded.

eonmorig
is an end-of-message limit pointer beyond which the expansion cannot go.

comp_dn
is a pointer to the first byte of the compressed name.

exp_dn
is a pointer to a buffer of size **length** that receives the expanded domain name.

dn_expand is part of the resolver. The resolver is a set of routines that provide a programming interface for communicating with Internet name servers. **dn_expand** translates domain names from the compressed format used by name servers to conventional character string format. In the interests of greater portability, the SAS/C version of **dn_expand** performs ASCII-to-EBCDIC translation of **exp_dn**.

For information on the UNIX programming interface and Internet name servers, refer to "The Domain Name System" and "The Socket Interface" in *Internetworking with TCP/IP, Volume I*.

RETURN VALUE

If successful, **dn_expand** returns the size of the compressed domain name. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

dn_expand is available on most versions of the UNIX operating system.

IMPLEMENTATION

The SAS/C version of **dn_expand** is a direct port from the BSD UNIX Socket Library. The ASCII-to-EBCDIC translation feature is the only change.

RELATED FUNCTIONS

dn_comp, **res_mkquery**

endhostent Closes a Host File or TCP Connection

SYNOPSIS

```
#include <netdb.h>

void endhostent(void);
```

DESCRIPTION

endhostent closes a host file or TCP connection. **endhostent** is a combination of the host file and resolver versions of the BSD UNIX Socket Library **endhostent**.

In some instances, when the resolver is used to look up the host name, a virtual circuit (that is, a TCP connection) is used to communicate with the name server, based on the **RESOLVEVIA** statement in the TCPIP.DATA file or the **RES_USEVC** resolver option specified by your program. In these cases, you can use the **RES_STAYOPEN** resolver option to maintain the connection with the name server between queries. **endhostent** closes this connection. (See "Bitwise OR Options" on page 15-12 for information about **RES_USEVC** and **RES_STAYOPEN**.)

In other instances, the host file is used as a source of host names. If the host file is opened with a **sethostent** call and the **stayopen** parameter is a nonzero value, **endhostent** closes the host file.

Refer to Chapter 17, "Network Administration" on page 17-1 for information on naming host files and the logic that determines whether the host file or the resolver is used for looking up names.

PORTABILITY

The logic that determines whether to use the host file or the resolver is not uniform across environments. At the source code level, however, **endhostent** is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

endhostent is a combination of the host file and resolver versions of the BSD UNIX Socket Library **gethostbyaddr**.

RELATED FUNCTIONS

sethostent, **gethostent**, **gethostbyname**

endnetent Closes the Network File

SYNOPSIS

```
#include <netdb.h>

void endnetent(void);
```

DESCRIPTION

endnetent closes the network file, that is, a file with the same format as **/etc/networks** in the UNIX environment. Refer to Chapter 17, “Network Administration” on page 17-1 for information on naming this file for your system.

PORTABILITY

endnetent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

This routine is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getnetbyaddr, **getnetbyname**, **getnetent**, **setnetent**

endprotoent Closes the Protocol File

SYNOPSIS

```
#include <netdb.h>

void endprotoent(void);
```

DESCRIPTION

endprotoent closes the protocol file, that is, a file with the same format as **/etc/protocols** in the UNIX environment. Refer to Chapter 17, “Network Administration” on page 17-1 for information on naming this file for your system.

PORTABILITY

endprotoent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

This routine is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getprotobynumber, **getprotobyname**, **getprotoent**, **setprotoent**

endrpcent Closes the /etc/rpc Protocol File

SYNOPSIS

```
#include <netdb.h>

int endrpcent(void);
```

DESCRIPTION

endrpcent closes the protocol file, that is, a file with the same format as **/etc/rpc** in the UNIX environment. Refer to Chapter 17, "Network Administration" on page 17-1 for information on naming this file for your system.

PORTABILITY

getrpcent is portable to other systems that support Sun RPC 4.0.

IMPLEMENTATION

This function is built from the Sun RPC 4.0 distribution.

RELATED FUNCTIONS

getrpcbyname, **getrpcbynumber**, **getrpcent**, **setrpcent**

endservent Closes the Services File

SYNOPSIS

```
#include <netdb.h>

void endservent(void);
```

DESCRIPTION

endservent closes the services file, that is, a file with the same format as **/etc/services** in the UNIX environment. Refer to Chapter 17, "Network Administration" on page 17-1 for information on naming this file for your system.

PORTABILITY

endservent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

This routine is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getservent, **setservent**, **getservbyname**, **getservbyport**

fcntl Controls Socket Operating Characteristics


SYNOPSIS

```
#include <sys/types.h>
#include <fcntl.h>

int fcntl(int filedes, int action, argument);
```

DESCRIPTION

Refer to the description of **`fcntl`** in *SAS/C Library Reference, Third Edition, Volume 2, Release 6.00* for a description of **`fcntl`** and the operating characteristics of sockets.

getclientid Gets the Calling Application Identifier

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int getclientid(int domain, struct clientid *clientid);
```

DESCRIPTION

getclientid gets the identifier of the calling application. **domain** is **AF_INET**. **clientid** is a pointer to the **clientid** structure, which is filled on return from the call. **getclientid** is used in the **givesocket** and **takesocket** calls, which enable cooperative processes to pass socket descriptors to each other.

Note: **getclientid** is only supported with non-integrated sockets.

RETURN VALUE

If **getclientid** succeeds, it returns a **0**. Otherwise, it returns a **-1** and sets **errno** to indicate the type of error.

PORTABILITY

getclientid is not portable to UNIX operating systems. On a UNIX operating system, sockets can be transferred from parent to child processes when the child process has been created via the **fork** system call.

EXAMPLE

In this example, **getclientid** returns the client ID from TCP/IP.

```
#include <stddef.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

   /* These three application routines implement communication  */
   /* between the parent task and this task. They use a means   */
   /* other than sockets to communicate.                        */
fromparent(void *, size_t);
toparent(void *, size_t);
postparent(void);

   /* This routine receives a socket from its parent            */
   /* task and store the descriptor into the integer pointed    */
   /* to by "s".                                                */
recvsock(int *s)
{
   struct clientid id;

      /* Get the clientid from TCP/IP.                          */
   if (getclientid (AF_INET, &id)==-1) {
      perror ("Can't get client ID");
      return -1;
   }
```

getclientid Gets the Calling Application Identifier

(continued)

```
      /* Pass clientid to parent.                              */
   toparent(&id,sizeof(id));

      /* Get socket descriptor number from parent.             */
   fromparent(s, sizeof(*s));

      /* Take socket from parent.                              */
   if (takesocket(&id, *s)==-1) {
      perror ("takesocket failed");
      return -1;
   }

      /* Tell parent that takesocket is completed.             */
   postparent();
   return 0;
}
```

RELATED FUNCTIONS

givesocket, **takesocket**

getdtablesize Gets Descriptor Table Size

SYNOPSIS

```
#include <sys/param.h>
int getdtablesize (void);
```

DESCRIPTION

getdtablesize returns the maximum number of HFS files and sockets that may be opened. In a system without OpenEdition, **getdtablesize** returns the maximum file descriptor number that can be returned for a socket.

RETURN VALUE

getdtablesize returns the maximum number of open HFS files and sockets if it is successful, and it returns a **-1** if it is not successful.

EXAMPLE

This example uses **getdtablesize** to determine the maximum number of sockets that can be opened and allocates storage for file descriptor sets large enough to accomodate this maximum. The file descriptor sets can be later passed to select.

Note that when file descriptor sets are allocated in this fashion, the **FD_ZERO** macro in **<sys/types.h>** should not be used as it assumes a fixed size for these sets.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>

main()
{
   int maxsockets;
   fd_set *readset, *writeset, *exceptset);
   int ready;

      /* get maximum number of sockets                                  */
   maxsockets = getdtablesize();
   readset = calloc(1, (maxsockets+7)/8);
   writeset = calloc(1, (maxsockets+7)/8);
   exceptset = calloc(1, (maxsockets+7)/8);

      /* allocate storage for fd sets (8 bits per byte)                 */
   .
   .
   .
   ready = select(maxsockets, readset, writeset, exceptset, NULL);

      /* wait for socket activity                                       */
   .
   .
   .
}
```

getdtablesize Gets Descriptor Table Size
(continued)

RELATED FUNCTIONS

`sysconf`

SEE ALSO

Chapter 3, "I/O Functions," in *SAS/C Library Reference, Volume 1.*

Chapter 2, "Function Categories," in *SAS/C Library Reference, Volume 1.*

gethostbyaddr Gets Host Information by Address

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

DESCRIPTION

Given the address of a host, **gethostbyaddr** returns a **hostent** structure containing the host's name and other information. This structure is typically used to obtain the name of the host from the **h_name** field. Refer to "**<netdb.h>**" on page 15-8 for details on the **hostent** structure. For TCP/IP, **addr** should point to **struct in_addr** or an unsigned long integer in network byte order, **len** is normally the **sizeof(struct in_addr)**, and **type** should be **AF_INET**.

Host information is found either through the resolver or in your system's equivalent of the **/etc/hosts** file. Refer to "**gethostbyname** and Resolver Configuration" on page 17-4 for a description of the logic that determines how the host address is found.

RETURN VALUE

If **gethostbyaddr** succeeds, it returns a host address. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **gethostbyaddr** returns points to a static structure within the library. You must copy the information from this structure before you make further **gethostbyname**, **gethostbyaddr**, or **gethostent** calls.

PORTABILITY

The logic that determines whether to use the host file or the resolver is not uniform across environments. At the source code level, however, **gethostbyaddr** is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C implementation of **gethostbyaddr** is a combination of the host file and resolver versions of the BSD UNIX Socket Library **gethostbyaddr** function.

RELATED FUNCTIONS

gethostbyname, **gethostent**, **sethostent**

gethostbyname Gets Host Information by Name

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

DESCRIPTION

Given the name of a host, **gethostbyname** returns a pointer to the **hostent** structure containing the host's IP address and other information. Refer to "<netdb.h>" on page 15-8 for details on the **hostent** structure. This structure is typically used to find the previous address of the host via the **h_addr** field. Host information is found either through the resolver or in your system's equivalent of the **/etc/hosts** file. Refer to "gethostbyname and Resolver Configuration" on page 17-4 for a description of the logic that determines how the host name is found.

RETURN VALUE

If **gethostbyname** succeeds, it returns a pointer to a host name. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **gethostbyname** returns points to a static structure within the library. You must copy the information from this structure before you make further **gethostbyname**, **gethostbyaddr**, or **gethostent** calls.

PORTABILITY

The logic that determines whether to use the host file or the resolver is not uniform across environments. At the source code level, however, **gethostbyname** is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C implementation of **gethostbyname** is a combination of the host file and resolver versions of the BSD UNIX Socket Library **gethostbyname** function.

gethostbyname Gets Host Information by Name

(continued)

EXAMPLE

This program uses the socket call, **gethostbyname** to return an IP address that corresponds to the supplied hostname. **gethostbyname** will determine if a nameserver or local host tables are being used for name resolution. The answer is returned in the hostent structure, **hp** and then printed. The local host must be properly configured for name resolution by either a nameserver or host tables.

```
#include <sys/types.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>

main(int argc, char *argv[])
{
   struct hostent *hp;
   struct in_addr ip_addr;

   /* Verify a "hostname" parameter was supplied */
   if (argc <1 || *argv[1] == '\0')
      exit(EXIT_FAILURE);

/* call gethostbyname() with a host name. gethostbyname() returns a */
/* pointer to a hostent struct or NULL.                             */
   hp = gethostbyname(argv[1]);

   if (!hp) {
      printf("%s was not resolved\n",argv[1]);
      exit(EXIT_FAILURE);
   }

/* move h_addr to ip_addr. This enables conversion to a form        */
/* suitable for printing with the inet_ntoa() function.             */

   ip_addr = *(struct in_addr *)(hp->h_addr);
   printf("Hostname: %s, was resolved to: %s\n",
           argv[1],inet_ntoa(ip_addr));

   exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

gethostbyaddr, **herror**, **sethostent**

gethostent Gets the Next Entry in the Host File

SYNOPSIS

```
#include <netdb.h>

struct hostent *gethostent(void);
```

DESCRIPTION

gethostent returns the next sequential entry in the host file.

RETURN VALUE

If **gethostent** succeeds, it returns a pointer to the **hostent** structure. Refer to "<netdb.h>" on page 15-8 for details on the **hostent** structure. A null pointer indicates an error occured or there were no more nework entries. If the resolver and the name server are in use, **gethostent** returns **NULL**.

CAUTION

The value that **gethostent** returns points to a static structure within the library. You must copy the information from this structure before you make further **gethostbyname**, **gethostbyaddr**, or **gethostent** calls.

PORTABILITY

The logic that determines whether to use the host file or the resolver is not uniform across environments. At the source code level, however, **gethostent** is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C implementation of **gethostent** is a combination of the host file and resolver versions of the BSD UNIX Socket Library **gethostent** function.

EXAMPLE

This program demonstrates the socket calls: **gethostent**, **endhostent**, and **sethostent**. The local host must be configured to use hosts tables for name resolution, *prefix*.ETC.HOSTS; where *prefix* is described in Chapter 17, "Network Administration" on page 17-1.

```
#include <sys/types.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1

static void prthost(struct hostent *h);

main(int argc, char *argv[])
```

gethostent Gets the Next Entry in the Host File
(continued)

```
{
   struct hostent *h;

/* sethostent() opens the prefix.ETC.HOSTS file, or when using a  */
/* nameserver, opens a TCP connection to the nameserver.          */

   sethostent(TRUE);

/* gethostent() reads the next sequential entry in the            */
/* prefix.ETC.HOSTS file. It returns a pointer to a "hostent"     */
/* structure.                                                     */

   while (h=gethostent())
      prthost(h);

/* endhostent() closes the prefix.ETC.HOSTS file, or the          */
/* connection to the nameserver.                                  */

   endhostent();
   exit(EXIT_SUCCESS);
}

/* prthost() prints the information returned by gethostent()      */
/* from the hostent structure.                                    */

static void prthost(struct hostent *h)
{
   char **p;

   /* Print primary name and aliases. */
   printf("\nname: %s\n",h->h_name);
   for (p=h->h_aliases; *p; p++)
      printf("alternate name: %s\n",*p);

   /* Handle unexpected situations gracefully. */
   if (h->h_addrtype != AF_INET) {
      printf("Not an internet address.\n");
      return;
   }
   if (h->h_length != sizeof(struct in_addr)) {
      printf("Invalid length: %d.\n",h->h_length);
      return;
   }

   /* Print the primary address and any alternates.  */
   for (p=h->h_addr_list; *p; p++) {
      printf("%s address: %s\n",
             p==h->h_addr_list ? "primary " : "alternate ",
             inet_ntoa((*(struct in_addr *)*p)) );
   }
}
```

gethostent Gets the Next Entry in the Host File
(continued)

RELATED FUNCTIONS

`endhostent`, **`gethostbyname`**, **`sethostent`**

gethostid Gets the Local Host's Internet Address

SYNOPSIS

```
#include <netdb.h>

unsigned long gethostid(void);
```

DESCRIPTION

gethostid gets the 32-bit Internet address for the local host.

RETURN VALUE

gethostid returns the Internet address or **-1** (**0xFFFFFFFF**), which is the value of the macro identifier **INADDR_NONE** in the **<netinet/in.h>** header file.

PORTABILITY

gethostid calls are not necessarily portable to all systems. In particular, the return value may not be the IP address. However, **gethostid** is portable to many other environments, including most UNIX systems, that implement BSD sockets. These other socket library implementations do not require the inclusion of the **<netdb.h>** header file. The SAS/C **<netdb.h>** header file contains **#pragma map** statements to create unique eight-character identifiers for the MVS and CMS linking utilities. To reduce incompatibilities caused by failure to include **<netdb.h>** in existing source code, a **#pragma map** statement for this function is also available in **<sys/types.h>**.

IMPLEMENTATION

The value that **gethostid** returns is assigned by the local host's TCP/IP software, which must be running in order for **gethostid** to succeed.

EXAMPLE

This program uses the socket call, **gethostid** to return the 32-bit internet address for the local host. The local host must have an operational TCPIP stack.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

main()
{
   struct in_addr in;

/* gethostid() returns the 32-bit internet address as an    */
/* unsigned long.                                           */
```

gethostid Gets the Local Host's Internet Address
(continued)

```
   in.s_addr = gethostid();
   if (in.s_addr == INADDR_NONE) {
      perror("gethostid failed");
      return EXIT_FAILURE;
   }

/* convert the unsigned long to a string in dotted decimal  */
/* and print.                                                */

   printf("Local Host IP Address is: %s\n",inet_ntoa(in));
   return EXIT_SUCCESS;
}
```

RELATED FUNCTIONS

gethostname

gethostname Gets the Host Name

SYNOPSIS

```
int gethostname(char *name, int namelen);
```

DESCRIPTION

gethostname returns the host name for the current processor. **namelen** is the size of the **name** array. The returned host name is null-terminated unless there is not enough space in **namelen**.

RETURN VALUE

If **gethostname** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

gethostname is portable to other environments, including most UNIX systems, that implement BSD sockets. These other socket library implementations do not require the inclusion of the **<netdb.h>** header file. The SAS/C **<netdb.h>** header file contains **#pragma map** statements to create unique eight-character identifiers for the MVS and CMS linking utilities.

IMPLEMENTATION

The value that **gethostname** returns is assigned by the local host's TCP/IP software, which must be running in order for **gethostid** to succeed.

PORTABILITY

gethostname is portable to other environments, including most UNIX systems, that implement BSD sockets. To reduce incompatibilities caused by failure to include **<netdb.h>** in existing source code, a **#pragma map** statement for this function is also available in **<sys/types.h>**.

EXAMPLE

This program uses the socket call, **gethostname** to return the hostname as defined to the local host. The local host must have an operational TCPIP stack.

```
#include <stdlib.h>
#include <netdb.h>
#include <stdio.h>

main()
{
   char buf[128];

/* gethostname returns the hostname in the buf array. If       */
/* gethostname() succeeds it returns a 0, otherwise a -1 and   */
/* errno is set accordingly.                                   */
```

gethostname Gets the Host Name

(continued)

```
    if (gethostname(buf,sizeof(buf))) {
       perror("Can't retrive host name");
       return EXIT_FAILURE;
    }
    printf("%s\n",buf);
    return EXIT_SUCCESS;
}
```

RELATED FUNCTIONS

gethostbyname, **gethostid**

GETLONG, _getlong Gets a Long Integer from a Character Buffer

SYNOPSIS

```
#include <sys/types.h>
#include <arpa/nameser.h>

GETLONG(l_int, msgp)

u_long _getlong(const u_char *msgp);
```

DESCRIPTION

The **GETLONG** macro and **_getlong** function extract an unsigned long integer **l_int** from a character buffer addressed by **msgp**. The bytes of the extracted integer are assumed to have been stored in the buffer as four consecutive bytes, starting with the high-order byte. The **_getlong** function returns the value. The **GETLONG** macro requires that both arguments be lvalues. **mgsp** is advanced by four bytes. The extracted unsigned long value is assigned to **l_int**.

This routine is useful in resolver programming. For information on buffer formats for Internet name servers, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP*.

RETURN VALUE

_getlong returns the value of the unsigned long integer. **GETLONG** is syntactically a statement rather than an expression and, therefore, has no return value.

CAUTION

GETLONG evaluates its arguments more than once.

IMPLEMENTATION

The **GETLONG** macro is defined in the **<arpa/nameser.h>** header file.

RELATED FUNCTIONS

GETSHORT, **PUTLONG**, **PUTSHORT**

getnetbyaddr Gets Network Information by Address

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetbyaddr(long net, int type);
```

DESCRIPTION

Given a network address, specified by **net**, **getnetbyaddr** returns a pointer to the **netent** structure as defined in **<netdb.h>**. This structure typically is used to obtain the network name from the **n_name** field. The network address should be supplied in host byte order. For TCP/IP, **type** should be **AF_INET**. Refer to "<netdb.h>" on page 15-8 for details on the **netent** structure. The source of the data in the **netent** structure is the network file, that is, a file with the same format as the **/etc/networks** file on a UNIX operating system.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the network file.

RETURN VALUE

If **getnetbyaddr** succeeds, it returns a pointer to the **netent** structure. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **getnetbyaddr** returns points to a static structure within the library. You must copy the information from this structure before you make further **getnetbyname**, **getnetbyaddr**, or **getnetent** calls.

PORTABILITY

getnetbyaddr is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getnetbyaddr is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getnetent, **getnetbyname**, **setnetent**, **endnetent**

getnetbyname Gets Network Information by Name

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetbyname(const char *name);
```

DESCRIPTION

Given a network name, pointed to by the **name** argument, **getnetbyname** returns a pointer to the **netent** structure, as defined in **<netdb.h>**. Refer to "<netdb.h>" on page 15-8 for details on the **netent** structure. This structure is typically used to obtain the network address from the **n_net** field. The source of the data in this structure is the network file, that is, a file with the same format as the **/etc/networks** file on a UNIX operating system.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the network file.

RETURN VALUE

If **getnetbyname** succeeds, it returns a pointer to the **netent** structure. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **getnetbyname** returns points to a static structure within the library. You must copy the information from this structure before you make further **getnetbyname**, **getnetbyaddr**, or **getnetent** calls.

PORTABILITY

getnetbyname is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getnetbyname is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getnetbyaddr, **getnetent**, **setnetent**, **endnetent**

getnetent Gets the Next Network Information Structure

SYNOPSIS

```
#include <netdb.h>

struct netent *getnetent(void);
```

DESCRIPTION

Given a network name, **getnetent** returns a pointer to the next network entry in the **netent** structure as defined in **<netdb.h>**. The source of the data in this structure is the network file, that is, a file with the same format as the **/etc/networks** file on a UNIX operating system. Refer to "<netdb.h>" on page 15-8 for details on the **netent** structure.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the network file.

RETURN VALUE

If **getnetent** succeeds, it returns a pointer to the **netent** structure. A null pointer indicates an error occurred or there were no more network entries.

CAUTION

The value that **getnetent** returns points to a static structure within the library. You must copy the information from this structure before you make further **getnetbyname**, **getnetbyaddr**, or **getnetent** calls.

PORTABILITY

getnetent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getnetent is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getnetbyaddr, **getnetbyname**, **setnetent**, **endnetent**

getpeername Gets the Address of a Peer

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int getpeername(int s, void *addr, int *addrlen);
```

DESCRIPTION

getpeername stores the address of the peer that is connected to socket **s**. The **addr** and **addrlen** functions describe the buffer into which **getpeername** places the address of the new peer. **addr** points to a **sockaddr** structure or one of its derivatives, such as **sockaddr_in**, and **addrlen** points to an integer containing the size of the buffer in bytes. If the buffer size is not large enough to contain the address of the peer, the value of the address is not completely copied. No error is indicated in this situation. On return, the integer pointed to by **addrlen** is set to the length that was actually copied. For connected sockets, the address that is returned is the same as that returned by the **recvfrom** function.

RETURN VALUE

If **getpeername** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

getpeername is portable to other environments, including most UNIX systems, that implement BSD sockets.

EXAMPLE

In this example, **getpeername** returns the port and address of the peer program.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

f()
{
   int s;
   struct sockaddr_in peer;
   int peer_len;
   .
   .
   .
      /* We must put the length in a variable.              */
   peer_len = sizeof(peer);
```

getpeername Gets the Address of a Peer

(continued)

```
      /* Ask getpeername to fill in peer's socket address.  */
   if (getpeername(s, &peer, &peer_len) == -1) {
      perror("getpeername() failed");
      return -1;
   }

      /* Print it. The IP address is often zero because     */
      /* sockets are seldom bound to a specific local       */
      /* interface.                                         */
   printf("Peer's IP address is: %s\n", inet_ntoa(peer.sin_addr));
   printf("Peer's port is: %d\n", (int) ntohs(peer.sin_port));
   .
   .
   .
}
```

RELATED FUNCTIONS

accept, **bind**, **connect**, **getsockname**, **recvfrom**

getprotobyname Gets Protocol Information by Name

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotobyname(char *name);
```

DESCRIPTION

Given the null-terminated name of a protocol, **getprotobyname** returns a pointer to the **protoent** structure defined in **<netdb.h>**. This structure is typically used to obtain the number for the protocol from the **p_proto** field. Refer to "<netdb.h>" on page 15-8 for details on the **protoent** structure. The source of the data in this structure is the protocols file, that is, a file with the same format as the **/etc/protocols** file on a UNIX operating system. Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the protocols file.

RETURN VALUE

If **getprotobyname** succeeds, it returns a pointer to the **protoent** structure. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **getprotobyname** returns points to a static structure within the library. You must copy the information from this structure before you make further **getprotobyname**, **getprotobynumber**, or **getprotoent** calls.

PORTABILITY

getprotobyname is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getprotobyname is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

endprotoent, **setprotoent**, **getprotobynumber**, **getprotoent**

getprotobynumber Gets Protocol Information by Number

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotobynumber(int proto);
```

DESCRIPTION

Given the number of a protocol, specified by **proto**, **getprotobynumber** returns a pointer to the **protoent** structure defined in **<netdb.h>** for the specified network protocol **proto**. Refer to "<netdb.h>" on page 15-8 for details on the **protoent** structure. This structure is typically used to obtain the name of the protocol from the **p_name** field. The source of the data in this structure is the protocols file, that is, a file with the same format as the **/etc/protocols** file on a UNIX operating system.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the protocols file.

RETURN VALUE

If **getprotobynumber** succeeds, it returns a pointer to the **protoent** structure. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **getprotobynumber** returns points to a static structure within the library. You must copy the information from this structure before you make further **getprotobyname**, **getprotobynumber**, or **getprotoent** calls.

PORTABILITY

getprotobynumber is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getprotobynumber is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

endprotoent, **setprotoent**, **getprotobyname**, **getprotoent**

getprotoent Gets Protocol Information

SYNOPSIS

```
#include <netdb.h>

struct protoent *getprotoent(void);
```

DESCRIPTION

Given a network name, **getprotoent** returns a pointer to the next network entry in the **protoent** structure defined in **<netdb.h>**. Refer to "<netdb.h>" on page 15-8 for details on the **protoent** structure. The source of the data in this structure is the protocols file, that is, a file with the same format as the **/etc/protocols** file on a UNIX operating system.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the protocols file.

RETURN VALUE

If **getprotoent** succeeds, it returns a pointer to the **protoent** structure. A null pointer indicates an error occurred or there were no more network entries.

CAUTION

The value that **getprotoent** returns points to a static structure within the library. You must copy the information from this structure before you make further **getprotobyname**, **getprotobynumber**, or **getprotoent** calls.

PORTABILITY

getprotoent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getprotoent is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

endprotoent, **setprotoent**, **getprotobyname**, **getprotobynumber**

getrpcbyname Returns `rpcent` Structure for an RPC Program Name

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcbyname(const char *name);
```

DESCRIPTION

Given the name of an RPC program pointed to by the **name** argument, **getrpcbyname** returns a pointer to the **rpcent** structure defined in **<netdb.h>**. This structure is typically used to obtain the number for the RPC program from the **r_number** field. Refer to “rpcent” on page 15-9 for details on the **rpcent** structure.

The source of the data in the **rpcent** structure is the protocols file, that is, a file with the same format as the **/etc/rpc** file on a UNIX operating system. Refer to “/etc/rpc” on page 17-3 for information on the logic used to determine the location of the protocols file.

RETURN VALUE

If **getrpcbyname** succeeds, it returns a pointer to the **rpcent** structure. A null pointer indicates an error or an end-of-file.

CAUTION

The value that **getrpcbyname** returns points to a static structure within the library. You must copy the information from this structure before you make further **getrpcbyname**, **getrpcbynumber**, or **getrpcent** calls.

PORTABILITY

getrpcbyname is portable to other systems that support Sun RPC 4.0.

IMPLEMENTATION

This function is built from the Sun RPC 4.0 distribution.

RELATED FUNCTIONS

endrpcent, **getrpcbynumber**, **getrpcent**, **setrpcent**

getrpcbynumber Returns the rpcent Structure for an RPC Program Number

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcbynumber(int prognum);
```

DESCRIPTION

Given the number of an RPC program, specified by **prognum**, **getrpcbynumber** returns a pointer to the **rpcent** structure, which is defined in **<netdb.h>**. This structure is typically used to obtain the name for the RPC program from the **r_name** field. Refer to "rpcent" on page 15-9 for details on the **rpcent** structure.

The source of the data in the **rpcent** structure is the protocols file, that is, a file with the same format as the **/etc/rpc** file on a UNIX operating system. Refer to "/etc/rpc" on page 17-3 for information on the logic used to determine the location of the protocols file.

RETURN VALUE

If **getrpcbynumber** succeeds, it returns a pointer to the **rpcent** structure. A null pointer indicates the network address was not found in the network file.

CAUTION

The value that **getrpcbynumber** returns points to a static structure within the library. You must copy the information from this structure before you make further **getrpcbyname**, **getrpcbynumber**, or **getrpcent** calls.

PORTABILITY

getrpcbynumber is portable to other systems that support Sun RPC 4.0.

IMPLEMENTATION

This function is built from the Sun RPC 4.0 distribution.

RELATED FUNCTIONS

endrpcent, **getrpcbyname**, **getrpcent**, **setrpcent**

getrpcent Returns the `rpcent` Structure

SYNOPSIS

```
#include <netdb.h>

struct rpcent *getrpcent(void);
```

DESCRIPTION

`getrpcent` returns a pointer to the **`rpcent`** structure, which is defined in **`<netdb.h>`**. The source of the data in this structure is the SUN RPC program numbers file, that is, a file with the same format as the **`/etc/rpc`** file on a UNIX operating system.

Refer to “/etc/rpc” on page 17-3 for information on the logic used to determine the location of the protocols file.

RETURN VALUE

If **`getrpcent`** succeeds, it returns a pointer to the **`rpcent`** structure. A null pointer indicates an error occurred or there were no more network entries.

CAUTION

The value that **`getrpcent`** returns points to a static structure within the library. You must copy the information from this structure before you make further **`getrpcbyname`**, **`getrpcbynumber`**, or **`getrpcent`** calls.

PORTABILITY

`getrpcent` is portable to other systems that support Sun RPC 4.0.

IMPLEMENTATION

This function is built from the Sun RPC 4.0 distribution.

RELATED FUNCTIONS

`endrpcent`, **`getrpcbyname`**, **`getrpcbynumber`**, **`setrpcent`**

getservbyname Gets Service Information by Name

SYNOPSIS

```
#include <netdb.h>

struct servent *getservbyname(const char *name, const char *proto);
```

DESCRIPTION

Given the name of a well-known service, pointed to by **name**, and a protocol string for accessing that service, pointed to by **proto**, **getservbyname** returns a pointer to the **servent** structure, which is defined in **<netdb.h>**. This structure is typically used to obtain the port for the service from the **serv_port** field. The source of the data in this structure is the services file, that is, a file with the same format as the **/etc/services** file on a UNIX operating system. Refer to "<netdb.h>" on page 15-8 for details on the **servent** structure.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the services file.

RETURN VALUE

If **getservbyname** succeeds, it returns a pointer to the **servent** structure. A null pointer indicates an error or an end-of-file.

CAUTION

The value that **getservbyname** returns points to a static structure within the library. You must copy the information from this structure before you make further **getservbyname**, **getservbyport**, or **getservent** calls.

PORTABILITY

getservbyname is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getservbyname is ported directly from the BSD UNIX Socket Library.

EXAMPLE

This program uses the socket call, **getservbyname** to obtain the structure, servent. In most cases the returned structure is used to obtain the port for the service. **getservbyname** reads the *prefix*.ETC.SERVICES file. The **prefix**.ETC.SERVICES file must be properly configures on the local host; where *prefix* is described in Chapter 17, "Network Administration" on page 17-1. The input parameters are case sensitive.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
```

getservbyname Gets Service Information by Name

(continued)

```
main(int argc, char *argv[])
{
   struct servent *serv;

   if (argc < 3) {
      puts("Incorrect parameters. Use:");
      puts("   gsbnm service-name protocol-name");
      return EXIT_FAILURE;
   }

   /* getservbyname() - opens the etc.services file and returns the */
   /* values for the requested service and protocol.                */

   serv = getservbyname(argv[1], argv[2]);
   if (serv == NULL) {
      printf("Service \"%s\" not found for protocol \"%s\"\n",
         argv[1], argv[2]);
      return EXIT_FAILURE;
   }

   /* Print it. */
   printf("Name: %-15s  Port: %5d    Protocol: %-6s\n",
            serv->s_name,ntohs(serv->s_port),serv->s_proto);
   return EXIT_SUCCESS;
}
```

RELATED FUNCTIONS

endservent, **setservent**, **getservent**, **getservbyport**

getservbyport Gets Service Information by Port

SYNOPSIS

```
#include <netdb.h>

struct servent *getservbyport(int port, const char *proto);
```

DESCRIPTION

Given the port of a well-known service, identified by the **port** argument, and the protocol string for accessing it, pointed to by **proto**, **getservbyport** returns a pointer to the **servent** structure, which is defined in **<netdb.h>**. This structure is typically used to obtain the name of the service from the **s_name** field. The source of the data in this structure is the services file, that is, a file with the same format as the **/etc/services** file on a UNIX operating system. Refer to "<netdb.h>" on page 15-8 for details on the **servent** structure.

Refer to "Search Logic" on page 17-1 for information on the logic used to determine the location of the services file.

RETURN VALUE

If **getservbyport** succeeds, it returns a pointer to the matching port number in the **servent** structure. A null pointer indicates an error occurred or there were no more network entries.

CAUTION

The value that **getservbyport** returns points to a static structure within the library. You must copy the information from this structure before you make further **getservbname**, **getservbyport**, or **getservent** calls.

PORTABILITY

getservbyport is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getservbyport is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

endservent, **setservent**, **getservent**, **getservbyname**

getservent Gets Next Entry in Services File

SYNOPSIS

```
#include <netdb.h>

struct servent *getservent(void);
```

DESCRIPTION

getservent returns a pointer to the next sequential entry in the services file, that is, a file with the same format as the **/etc/services** file on a UNIX operating system. Refer to “<netdb.h>” on page 15-8 for details on the **servent** structure.

Refer to “Search Logic” on page 17-1 for information on the logic used to determine the location of the services file.

RETURN VALUE

If **getservent** succeeds, it returns a pointer to the **servent** structure. A null pointer indicates an error or an end-of-file.

CAUTION

The value that **getservent** returns points to a static structure within the library. You must copy the information from this structure before you make further **getservbyname**, **getservbyport**, or **getservent** calls.

PORTABILITY

getservent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

getservent is ported directly from the BSD UNIX Socket Library.

EXAMPLE

This program demonstrates the socket calls: **endservent**, **getservent**, and **setservent**. GETSENT attempts to open a *prefix*.ETC.SERVICES file to obtain local configuration data; where *prefix* is described in Chapter 17, “Network Administration” on page 17-1.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define TRUE 1
```

getservent Gets Next Entry in Services File

(continued)

```
main()
{
   struct servent *serv;

   /* setservent() opens the ETC.SERVICES file.             */

   setservent(TRUE);

   /* getservent() sequentially reads the elements in the   */
   /* ETC.SERVICES file.                                    */

   while (serv = getservent()) {
     /* Print an entry. */
     printf("Name: %-15s  Port: %5d    Protocol: %-6s\n",
             serv->s_name,ntohs(serv->s_port),serv->s_proto);
   }

   /* endservent() closes the ETC.SERVICES file.            */

   endservent();
   return EXIT_SUCCESS;
}
```

RELATED FUNCTIONS

endservent, **setservent**, **getservbyname**, **getservbyport**

GETSHORT, _getshort

Gets a Short Integer from Character Buffer

SYNOPSIS

```
#include <sys/types.h>
#include <arpa/nameser.h>

GETSHORT(s_int, msgp)

u_short _getshort(const u_char *msgp);
```

DESCRIPTION

The **GETSHORT** macro and **_getshort** function extract an unsigned short integer **s_int** from a character buffer addressed by **msgp**. The bytes of the extracted integer are assumed to have been stored in the buffer as two consecutive bytes, starting with the high-order byte. The **_getshort** function returns the value. The **GETSHORT** macro requires that both arguments are lvalues. **mgsp** is advanced by two bytes. The extracted unsigned short value is assigned to **s_int**.

This routine is useful in resolver programming. For information on buffer formats for Internet name servers, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP*.

RETURN VALUE

_getshort returns the value of the unsigned short integer. **GETSHORT** is syntactically a statement rather than an expression, and therefore has no return value.

CAUTION

GETSHORT evaluates its arguments more than once.

IMPLEMENTATION

The **GETSHORT** macro is defined in the **<arpa/nameser.h>** header file.

RELATED FUNCTIONS

GETLONG, **PUTLONG**, **PUTSHORT**

getsockname Gets a Socket by Name

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockname(int s, void *addr, int *addrlen);
```

DESCRIPTION

getsockname stores the address that is bound to a specified socket **s** in the buffer pointed to by **addr**. The **addr** and **addrlen** functions describe the buffer into which **getsockname** places the address of the socket. **addr**, as specified in the previous **bind** call, should point to a **sockaddr** structure or one of its derivatives, such as **sockaddr_in**.

addrlen points to an integer containing the size of the buffer in bytes. If the buffer size is not large enough to contain the address of the socket, the value of the address is not completely copied. No error is indicated in this situation. On return, the integer pointed to by **addrlen** is set to the length that was actually copied. If the socket has not been bound to an address, only the **sa_family** field is meaningful; all other fields are set to **0**.

RETURN VALUE

If **getsockname** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

getsockname is portable to other environments, including most UNIX systems, that implement BSD sockets.

EXAMPLE

In this example **getsockname** returns the name of bound socket **s**.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

f()
{
   int s;
   struct sockaddr_in sa;
   int sa_len;
   .
   .
   .
```

getsockname Gets a Socket by Name
(continued)

```
      /* We must put the length in a variable.              */
   sa_len = sizeof(sa);
      /* Ask getsockname to fill in this socket's local     */
      /* address.                                           */
   if (getsockname(s, &sa, &sa_len) == -1) {
      perror("getsockname() failed");
      return -1;
   }

      /* Print it. The IP address is often zero beacuase    */
      /* sockets are seldom bound to a specific local       */
      /* interface.                                         */
   printf("Local IP address is: %s\n", inet_ntoa(sa.sin_add r));
   printf("Local port is: %d\n", (int) ntohs(sa.sin_port));
   .
   .
   .
}
```

RELATED FUNCTIONS

bind, **getpeername**

getsockopt Gets the Value of an Option

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int getsockopt(int s, int level, int optname, void *optval, int *optlen);
```

DESCRIPTION

getsockopt returns the value of an option associated with socket **s**. The **level** argument is the level of the option. The **optname** argument is the name of the option. The **optval** argument is a pointer to the buffer that receives the value of the requested option. As an input parameter, the integer pointed to by **optlen** should be set to the size of the **optval** buffer. On output, the integer pointed to by **optlen** is set to the size of the returned value. For cases in which **optval** points to an integer, a stored value of **0** indicates that the option is off. A nonzero returned value indicates that the option is on.

Most options are at the socket level. Pass **SOL_SOCKET** as the **level** parameter for these options:

SO_ERROR
gets the error status of the socket. The TCP/IP software maintains an **errno** value (the **SO_ERROR** field on a UNIX operating system) for each socket. The **SO_ERROR** option obtains and then clears this field, which is useful when checking for errors that occur between socket calls. In this case, **optval** should point to an integer.

SO_LINGER
gets the setting of the **linger** option. The **linger** option controls the behavior of the call to **close** when some data have not yet been sent. **optval** should point to a **struct linger**. If the value of the **l_onoff** field is **0**, **close** returns immediately. In this case, TCP/IP continues to attempt to deliver the data, but the program is not informed if delivery fails. If the value of the **l_onoff** field is nonzero, the behavior of the **close** call depends on the value of the **l_linger** field. If this value is **0**, unsent data are discarded at the time of the **close**. If the value of **l_linger** is not **0**, the **close** call blocks the caller until there is a timeout or until all data are sent. The **l_linger** field indicates the length of the desired timeout period. Some TCP/IP implementations may ignore the value of the **l_linger** field. **s** must be a stream socket.

SO_OOBINLINE
receives out-of-band data inline, that is, without setting the **MSG_OOB** flag in **recv** calls. **optval** should point to an integer. **s** must be a stream socket. Refer to "Out-of-Band Data" in Chapter 6, "Berkeley Sockets," in *UNIX Network Programming* for information on out-of-band data.

SO_REUSEADDR
allows local port address to be reused. **optval** should point to an integer. **s** must be a stream socket.

SO_TYPE
gets type of socket: **SOCKET_STREAM**, **SOCK_RAW**, or **SOCK_DGRAM**. The **optval** pointer should point to an integer.

getsockopt Gets the Value of an Option
(continued)

The option level **IPPROTO_TCP** is available for one option. The **TCP_NODELAY** option indicates that TCP's normal socket buffering should not be used on the socket. The **TCP_NODELAY** option is not operative; it is supported for source code compatibility. The **<netinet/in.h>** and **<netinet/tcp.h>** headers files are required for this option.

RETURN VALUE

If **getsockopt** succeeds, it returns a **0**. Otherwise, it returns a **-1** and sets **errno** to indicate the type of error.

PORTABILITY

getsockopt is portable to other environments, including most UNIX systems, that implement BSD sockets. The supported options may vary depending on the system, and the options described previously may be supported differently. Additional options may be supported by a specific TCP/IP software product. Refer to your software documentation for details.

EXAMPLE

In this example **getsockopt** gets the type of socket.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

main()
{
   int optlen, gs, socktype, s;

      /* Create a datagram socket. */
   s = socket(AF_INET, SOCK_DGRAM, 0);
   if (s == -1) {
      perror("Socket not created");
      return EXIT_FAILURE;
   }

      /* Ask for the socket type. */
   optlen = sizeof(socktype);
   gs = getsockopt (s, SOL_SOCKET, SO_TYPE, &socktype, &optlen);
   if (gs == -1) {
      perror("getsockopt failed");
      return EXIT_FAILURE;
   }
```

getsockopt Gets the Value of an Option
(continued)

```
   /* Print socket type. */
  switch (socktype)
  {
  case SOCK_STREAM:
     puts("Stream socket.\n");
     break;
  case SOCK_DGRAM:
     puts("Datagram socket.\n");
     break;
  case SOCK_RAW:
     puts("Raw socket.\n");
     break;
  default:
     puts("Unknown socket type.\n");
     break;
  }
  return EXIT_SUCCESS;
}
```

RELATED FUNCTIONS

bind, **close**, **fcntl**, **ioctl**, **recv**, **setsockopt**, **socket**

givesocket Gives a Socket to Another Process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int givesocket(int s, const struct clientid * clientid);
```

DESCRIPTION

givesocket specifies that socket **s** is available to another process. **s** must be a connected stream socket. After a successful **givesocket** call, the donor process, which is often a concurrent server, passes its client ID **clientid** and the socket descriptor for **s** to the receiving process. The donor process must have already obtained the client ID of the receiving process.

To pass a socket, the donor program calls **givesocket** with the **clientid** structure filled in as follows:

Field	Description
domain	**AF_INET**
name	Receiving program's address space name, left justified and padded with blanks
subtaskname	blanks
reserved	binary zeros

The receiving process then completes the operation by issuing a **takesocket** call. The mechanism for exchanging client IDs and socket descriptor values is the responsibility of the two processes and is not accomplished by either the **givesocket** or the **takesocket** call.

If **givesocket** is successful, **s** is not available for any other calls until a **close** is issued for **s**. Do not issue a **close** call until the other application has successfully completed a **takesocket** call for **s**; otherwise the connection is reset.

The **select** or **selectecb** functions can be used to wait on an exception condition for a given socket. An exception condition is raised on a socket when the receiver has taken the socket with the **socket** function. When the exception is raised, the donor program can close the socket.

Note: **givesocket** is only supported with non-integrated sockets.

RETURN VALUE

If **givesocket** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

givesocket Gives a Socket to Another Process
(continued)

PORTABILITY

givesocket is not portable to UNIX operating systems. On UNIX operating systems, sockets are typically transferred from parent to child processes as a side effect of the **fork** system call.

RELATED FUNCTIONS

getclientid, **select**, **selectecb**, **takesocket**

herror Prints a Host Error Message

SYNOPSIS

```
#include <netdb.h>

void herror(const char *string);
```

DESCRIPTION

herror writes an error message to **stderr** describing a failure in **gethostbyname** or **gethostbyaddr**. If **string** is not **NULL**, it is written first, followed by a colon, a space, and the error message corresponding to the **h_errno** value.

h_errno values and **herrno** text are as follows:

h_errno value	**herror text**
HOST_NOT_FOUND	Host not found
TRY_AGAIN	Temporary failure, try later
NO_RECOVERY	Nameserver failure
NO_DATA	No data of this type associated with this name
NO_ADDRESS	No address of this type associated with this name

PORTABILITY

herror is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

gethostbyaddr, **gethostbyname**

htoncs Converts an EBCDIC Character to ASCII

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

int htoncs(int hostchar);
```

DESCRIPTION

htoncs converts a single EBCDIC character **hostchar** to an ASCII character. High-order bits will be ignored in the input parameter. The name of this function, an abbreviation of "host to network character set," is analogous to the **htonl** and **htons** functions. ASCII is the predominant character set for text on TCP/IP networks, while EBCDIC is the predominant character set for text on MVS and CMS systems.

The **htoncs** function is provided to facilitate the writing of TCP/IP programs on MVS and CMS systems. You may find that other EBCDIC-to-ASCII translation routines are better suited to your requirements.

RETURN VALUE

htoncs returns the ASCII character corresponding to the input argument interpreted as an EBCDIC character.

PORTABILITY

htoncs is not portable; character set translation is seldom required in other environments.

IMPLEMENTATION

Refer to member L$USKCS in SASC.SOURCE (MVS) or LSU MACLIB (CMS) for the translate table. This routine may be modified by the user.

RELATED FUNCTIONS

htonl, **htons**, **ntohcs**, **ntohl**, **ntohs**

htonl Converts a Long Integer from Host to Network Order

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long htonl(unsigned long hostlong);
```

DESCRIPTION

The **htonl** macro converts an unsigned long integer **hostlong** from host byte order to network byte order. On an IBM 370, this is a null macro since IBM 370 byte ordering is big endian, as is network byte ordering.

RETURN VALUE

htonl returns the converted value.

CAUTION

There is also an **htonl** function. Be sure to include the proper header files so that the **htonl** macro is always used.

PORTABILITY

htonl is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The **htonl** macro is defined in the **<netinet/in.h>** header file.

RELATED FUNCTIONS

htoncs, **htons**, **ntohcs**, **ntohs**, **ntohl**

htons Converts a Short Integer from Host to Network Order

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned short htons(unsigned short hostshort);
```

DESCRIPTION

The **htons** macro converts an unsigned short integer **hostshort** from host byte order to network byte order. On an IBM 370, this is a null macro since IBM 370 byte ordering is big endian, as is network byte ordering.

RETURN VALUE

htons returns the converted value.

CAUTION

There is also an **htons** function. Be sure to include the proper header files so that the **htons** macro is always used.

PORTABILITY

htons is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The **htons** macro is defined in the **<netinet/in.h>** header file.

RELATED FUNCTIONS

htoncs, **htonl**, **ntohcs**, **ntohs**, **ntohl**

inet_addr Interprets an Internet Address

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_addr(const char *cp);
```

DESCRIPTION

inet_addr interprets a null-terminated character string, pointed to by **cp**, that represents numbers in the Internet standard dotted decimal notation and returns a corresponding Internet address. The dotted decimal string can contain up to four components. All but the last components are placed in succeeding bytes, beginning with the high-order byte. The last component fills all remaining bytes. This dotted decimal notation takes one of the following forms:

a.b.c.d

Each section is assigned, from left to right, to the four bytes of an Internet address.

a.b.c

The last section is interpreted as a 16-byte quantity and placed in the rightmost two bytes of the network address. In this way, you can specify Class B network addresses as "128.net.host".

a.b

The last part is interpreted as a 24-bit quantity and placed in the rightmost three bytes of the network address. In this way, you can specify Class A network addresses as "net.host".

a.

This value is stored directly in the network address without rearranging any bytes.

The numbers supplied in dotted decimal notation can be decimal, octal, or hexadecimal, as specified in the C language. In other words, a leading **0x** or **0X** implies hexadecimal notation; a leading **0** implies octal notation. Otherwise, the number is interpreted as decimal.

The Internet address is returned in network byte order.

RETURN VALUE

If **inet_addr** is successful, it returns the address in network byte order. Otherwise, it returns a **-1UL** (**0xFFFFFFFF**), and sets **errno** to indicate the type of error. INADDR_NONE is the symbolic name for the **-1UL** value returned by **inet_addr** when the input is valid.

PORTABILITY

inet_addr is portable to other environments, including most UNIX systems, that implement BSD sockets. On some systems, this routine may return the type **struct in_addr**.

inet_addr Interprets an Internet Address
(continued)

IMPLEMENTATION

The SAS/C version of **inet_addr** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

inet_lnaof, **inet_makeaddr**, **inet_netof**, **inet_network**, **inet_ntoa**

inet_lnaof Determines a Local Network Address

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_lnaof(struct in_addr in);
```

DESCRIPTION

inet_lnaof separates the elements in an Internet host address, specified by **in**, and returns the local network address.

The host address is in network byte order because it is contained in a **struct in_addr**.

RETURN VALUE

inet_lnaof returns the network address in host byte order.

PORTABILITY

inet_lnaof is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C version of **inet_lnaof** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

inet_addr, **inet_makeaddr**, **inet_netof**, **inet_network**, **inet_ntoa**

inet_makeaddr Constructs an Internet Address

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

struct in_addr inet_makeaddr(int net, int lna);
```

DESCRIPTION

inet_makeaddr constructs an Internet address from an Internet network number **net** and a local network address **lna**. Both the Internet network number and the local network address are specified in host byte order.

RETURN VALUE

inet_makeaddr returns the Internet address in network byte order.

PORTABILITY

inet_makeaddr is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C version of **inet_makeaddr** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

inet_addr, **inet_lnaof**, **inet_netof**, **inet_network**, **inet_ntoa**

inet_netof Determines the Network Number

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_netof(struct in_addr in);
```

DESCRIPTION

inet_netof separates the elements in an Internet host address **in** and returns the network number. The host address is specified in network byte order because it is contained in a **struct in_addr**.

RETURN VALUE

inet_netof returns the network number in host byte order.

PORTABILITY

inet_netof is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C version of **inet_netof** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

inet_addr, **inet_lnaof**, **inet_makeaddr**, **inet_network**, **inet_ntoa**

inet_network Interprets an Internet Network Number

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

unsigned long inet_network(const char *cp);
```

DESCRIPTION

inet_network interprets a null-terminated character string, pointed to by **cp**, that represents numbers in the Internet standard dotted decimal notation and returns that string as an Internet network number of up to four components. Each component is assigned to a byte of the result. If there are fewer than four components, high-order bytes have a value of **0**.

The numbers supplied in dotted decimal notation can be decimal, octal, or hexadecimal, as specified in the C language. In other words, a leading **0x** or **0X** implies hexadecimal notation; a leading **0** implies octal notation. Otherwise, the number is interpreted as decimal.

RETURN VALUE

If **inet_network** is successful, it returns the network number. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error. **INADDR_NONE** is the symbolic name for the **-1** value returned by **inet_network** when the input is valid.

PORTABILITY

inet_network is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C version of **inet_network** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

inet_lnaof, **inet_makeaddr**, **inet_netof**, **inet_addr**, **inet_ntoa**

inet_ntoa Puts an Internet Address into Network Byte Order

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

DESCRIPTION

inet_ntoa takes an Internet address, **in**, and returns a pointer to a null-terminated string representing the address in dotted decimal notation. The dotted decimal string has four components. The host address is specified in network byte order because it is contained in a **struct in_addr**.

RETURN VALUE

If **inet_ntoa** is successful, it returns a pointer to the Internet address in dotted decimal notation.

CAUTION

The string value is contained in a static character array. You must copy this value if you plan to refer to it after a subsequent **inet_ntoa** call.

PORTABILITY

inet_ntoa is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The SAS/C version of **inet_ntoa** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

inet_addr, **inet_lnaof**, **inet_makeaddr**, **inet_network**, **inet_netof**

ioctl Controls Operating Characteristics of Socket Descriptors

SYNOPSIS

```
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <if.h>

int ioctl(int s, unsigned long cmd, void *data);
```

DESCRIPTION

ioctl controls or queries the operating characteristics of socket descriptor **s**. **data** points to the data that are associated with command **cmd**.

The following commands are valid:

FIONBIO
controls non-blocking I/O for socket descriptor **s**, and **data** points to an integer. If ***data** is **0**, **ioctl** clears non-blocking I/O. If ***data** is not **0**, **s** is set for non-blocking I/O. Calls that cannot complete immediately return a **-1** with **errno** set to **EWOULDBLOCK**.

FIONREAD
stores the number of readable bytes for **s**; **data** points to an integer, which is set to the number of readable characters for **s**.

SIOCATMARK
checks to see if **s** is pointing to out-of-band data. **data** points to an integer, which is set to **1** if **s** points to out-of-band data. Otherwise, ***data** is set to **0**. Use this option for TCP sockets where out-of-band data are delivered inline. Refer to "Out-of-Band Data" in Chapter 6, "Berkeley Sockets," in *UNIX Network Programming.*

SIOCGIFADDR
stores the network interface address. **data** points to an **ifreq** structure that is defined in **<if.h>**. The address is returned in the **if_addr** field.

SIOCGIFBRDADDR
stores the network interface broadcast address. **data** points to an **ifreq** structure that is defined in **<if.h>**. The address is returned in the **if_broadaddr** field.

SIOCGIFCONF
stores a network interface configuration list. **data** points to an **ifconf** structure that is defined in **<if.h>**. On input, the structure specifies a buffer into which the list is placed. The **ifc_buf** field should point to the buffer. The **ifc_len** field should contain its length in bytes.

SIOCGIFDSTADDR
stores the network destination interface address. **data** points to an **ifreq** structure that is defined in **<if.h>**. For point-to-point connections, this option stores the address of the remote interface or destination. The address is stored in the **if_dstadaddr** field.

ioctl Controls Operating Characteristics of Socket Descriptors
(continued)

SIOCGIFFLAGS
stores the network interface flags. **data** points to an **ifreq** structure that is defined in **<if.h>**. The flags provide information about the interface and its current state, such as whether the interface is point-to-point and whether it is currently up. The flags are stored in the **ifr_flags** field of the **ifreq** structure. The names of the flags begin with IFF; they are listed in the **<if.h>** header file.

SIOCGIFNETMASK
stores the network interface network mask. **data** points to an **ifreq** structure that is defined in **<if.h>**. The address is returned in the **if_addr** field.

Additional options may be supported by some TCP/IP software products. Refer to your software documentation for details.

Note: If integrated sockets are in use, the **ioctl** function simply calls the **w_ioctl** OpenEdition system call.

RETURN VALUE

If **ioctl** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

ioctl is portable to other environments, including most UNIX systems, that implement BSD sockets. Unlike the BSD UNIX **ioctl** call, the SAS/C version of **ioctl** controls only sockets.

RELATED FUNCTIONS

fcntl **getsockopt**, **setsockopt**

listen Indicates Socket Descriptor is Ready to Accept Requests

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int s, int backlog);
```

DESCRIPTION

listen indicates that the socket descriptor **s** is ready to accept incoming connection requests. **backlog** is an integer defining the maximum length of the queue of connection requests. The SOMAXCONN constant in the **<socket.h>** header file defines the maximum value for the **backlog** parameter. Currently, **SOMAXCONN** is **10**. The connections are accepted with **accept**. Servers typically use this call in preparation for service requests from clients.

RETURN VALUE

If **listen** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

listen is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

accept, **bind**, **connect**

ntohcs Converts ASCII Characters to EBCDIC

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

int ntohcs(int netchar);
```

DESCRIPTION

ntohcs converts a network ASCII character **netchar** to a host EBCDIC character. High-order bits will be ignored in input parameters. The name of this function, an abbreviation of "network to host character set," is analogous to the names of the **ntohl** and **ntohs** functions. ASCII is the standard character set for text on TCP/IP networks, while EBCDIC is the standard character set for text on MVS and CMS systems.

The **ntohcs** function is provided to facilitate the writing of TCP/IP programs on MVS and CMS systems. You may find that other ASCII-to-EBCDIC translation routines are better suited to your requirements.

RETURN VALUE

ntohcs returns the EBCDIC character corresponding to **netchar** interpreted as an ASCII character.

PORTABILITY

ntohcs is not portable; character set translation is seldom required in other environments.

IMPLEMENTATION

Refer to member L$USKCS in SASC.SOURCE (MVS) or LSU MACLIB (CMS) for the translate table. This routine may be modified by the user.

RELATED FUNCTIONS

htoncs, **htonl**, **htons**, **ntohl**, **ntohs**

ntohl Converts a Long Integer from Network to Host Byte Order

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned long ntohl(unsigned long netlong);
```

DESCRIPTION

The **ntohl** macro converts an unsigned long integer **netlong** from network byte order to host byte order. On an IBM 370, this is a null macro, since IBM 370 byte ordering is big endian, as is network byte ordering.

RETURN VALUE

ntohl returns the converted value.

CAUTION

There is also an **ntohl** function. Be sure to include the proper header files so that the **ntohl** macro is always used.

PORTABILITY

ntohl is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The **ntohl** macro is defined in the **<netinet/in.h>** header file.

RELATED FUNCTIONS

htoncs, **htonl**, **htons**, **ntohcs**, **ntohs**

ntohs Converts a Short Integer from Network to Host Byte Order

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

unsigned short ntohs(unsigned short netshort);
```

DESCRIPTION

The **ntohs** macro converts an unsigned short integer **netshort** from network byte order to host byte order. On an IBM 370, this is a null macro, since IBM 370 byte ordering is big endian, as is network byte ordering.

RETURN VALUE

ntohs returns the converted value.

CAUTION

There is also an **ntohs** function. Be sure to include the proper header files so that the **ntohs** macro is always used.

PORTABILITY

ntohs is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

The **ntohs** macro is defined in the **<netinet/in.h>** header file.

RELATED FUNCTIONS

htoncs, **htonl**, **htons**, **ntohcs**, **ntohl**

PUTLONG, putlong Puts a Long Integer into a Character Buffer

SYNOPSIS

```
#include <sys/types.h>
#include <arpa/nameser.h>

PUTLONG(ul, msgp)

int putlong(u_long ul, u_char *msgp);
```

DESCRIPTION

The **PUTLONG** macro and **putlong** function put an unsigned long integer **ul** into a character buffer addressed by **msgp**. The bytes of the integer are assumed to have been stored in the buffer as four consecutive bytes, starting with the high-order byte. The **putlong** function returns the value. The **PUTLONG** macro requires that both its arguments be lvalues. **mgsp** is advanced by four bytes. The value in **ul** is destroyed by the macro. These routines are useful in resolver programming. For information on buffer formats for Internet name servers, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP*.

RETURN VALUE

putlong returns the value of the unsigned long integer. **PUTLONG** is syntactically a statement rather than an expression and, therefore, has no return value.

CAUTION

PUTLONG evaluates its arguments more than once. The **PUTLONG** macro destroys its first argument.

IMPLEMENTATION

The **PUTLONG** macro is defined in the **<arpa/nameser.h>** header file.

RELATED FUNCTIONS

GETLONG, **GETSHORT**, **PUTSHORT**

PUTSHORT, putshort

Puts a Short Integer into a Character Buffer

SYNOPSIS

```
#include <sys/types.h>
#include <arpa/nameser.h>

PUTSHORT(u_sh, mgsp)

int putshort(u_short u_sh, u_char *msgp);
```

DESCRIPTION

The **PUTSHORT** macro and **putshort** function put an unsigned short integer **u_sh** into a character buffer addressed by **msgp**. The **PUTSHORT** macro requires that its second argument be an lvalue. **mgsp** is advanced by two bytes. This routine is useful in resolver programming. For information on buffer formats for Internet name servers, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP*.

RETURN VALUE

putshort returns the value of the unsigned short integer. **PUTSHORT** is syntactically a statement rather than an expression, and, therefore, has no return value.

CAUTION

PUTSHORT evaluates its arguments more than once.

IMPLEMENTATION

The **PUTSHORT** macro is defined in the **<arpa/nameser.h>** header file.

RELATED FUNCTIONS

GETLONG, **GETSHORT**, **PUTLONG**

readv Reads Data from a Socket Descriptor into an Array of Buffers

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <fcntl.h>

int readv(int s, struct iovec *iov, int iovcnt);
```

DESCRIPTION

readv reads data from socket or file descriptor **s** into the **iovcnt** buffers specified by the **iov** array. As with the **read** call, the socket must have been previously associated with a remote address via the **connect** system call. If there are no data, **readv** blocks the caller unless the socket is in non-blocking mode. The **iovec** structure is defined in **<sys/uio.h>**. Each **iovec** entry specifies the base address and length of an area in memory in which the data should be placed. **readv** completely fills one area before proceeding to the next area.

Note: Although **readv** is primarily used with sockets, it can also be used to read any file that can be accessed by the **read** function.

RETURN VALUE

If **readv** succeeds, it returns the number of bytes read into the buffer. If **readv** returns a **0**, the end-of-file has been reached. If **readv** fails, it returns a **-1**. It is common for **readv** to return a value less than the total number of bytes in the buffers. This is not an error.

CAUTION

readv is an atomic operation. With UDP, no more than one datagram can be read per call. If you are using datagram sockets, make sure that there is enough buffer space in the I/O vector to contain an incoming datagram.

PORTABILITY

readv is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

connect, **recv**, **recvfrom**, **recvmsg**, **write**, **writev**

recv Stores Messages from a Connected Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int s, void *buf, int len, int flags);
```

DESCRIPTION

recv receives messages on socket **s** and stores them in the buffer **buf** of length **len**. The socket must be connected or associated with a foreign peer via the **connect** function. If no messages are available, **recv** blocks the caller until a message arrives, unless the socket is set in non-blocking mode.

flags consists of the following:

MSG_OOB
processes a single byte of out-of-band data, if such data are present. If out-of-band data have been moved inline with the **SO_OOBINLINE** socket option, multiple bytes of data can be accessed, and the **MSG_OOB** option is not necessary.

MSG_PEEK
peeks at the incoming message. The data remain available for the next **recv** call.

read and **recv** behave identically if no flags have been set for **recv**.

RETURN VALUE

If **recv** succeeds, it returns the length of the message. If **recv** returns a **0**, it indicates that the connection is closed. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error. It is possible for **recv** to return fewer bytes than are specified by **len**. For example, this condition occurs if the number of bytes in an incoming datagram is less than **len**. This is not an error.

CAUTION

recv discards excess bytes if the datagram is larger than the specified buffer.

PORTABILITY

recv is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

connect, **getsockopt**, **recvfrom**, **recvmsg**, **send**, **sendmsg**, **sendto**, **setsockopt**

recvfrom Stores Messages from a Connected or Unconnected Socket Descriptor

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int recvfrom(int s, void *buf, int len, int flags, void *from,
             int *addrlen);
```

DESCRIPTION

recvfrom receives data from an unconnected or connected socket descriptor **s**. If **from** is nonzero, the source address of the message is stored in the buffer addressed by **from**. **addrlen** is the size of the buffer pointed to by **from**. **buf** points to the buffer receiving the message. **len** is the size of the message. **recvfrom** is particularly useful for unconnected datagram sockets.

If no messages are available, **recvfrom** blocks the call until a message arrives unless the socket is set in non-blocking mode. **flags** consists of the following:

MSG_OOB
processes a single byte of out-of-band data if data are present. If out-of-band data have been moved inline with the **SO_OOBINLINE** socket option, multiple bytes of data can be accessed, and the **MSG_OOB** option is not necessary.

MSG_PEEK
peeks at the incoming message. The data remain available for the next **recvfrom** call.

RETURN VALUE

If **recvfrom** succeeds, it returns the length of the message. If **recv** returns a **0**, it indicates that the connection is closed. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error. It is common for **recvfrom** to return fewer than the total number of bytes in the buffers. This is not an error.

CAUTION

recvfrom discards excess bytes if a datagram is larger than the specified buffer.

PORTABILITY

recvfrom is portable to other environments, including most UNIX systems, that implement BSD sockets.

recvfrom Stores Messages from a Connected or Unconnected Socket Descriptor
(continued)

EXAMPLE

In this example, **recvfrom** receives a datagram on socket descriptor **s** and places it in buffer **in**.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>

#define MAX_DGRAM_SIZE 256    /* Chosen by the application. */

f()
{
   int s;
   int b;
   struct sockaddr_in from;            /* Sender's address. */
   int fromlen;              /* Length of sender's address. */
   char in[MAX_DGRAM_SIZE] ;

      /* Open a datagram socket to receive the packet.         */
   s = socket(AF_INET, SOCK_DGRAM, 0);
   .
   .
   .
      /* Receive a datagram into the buffer.                   */
   fromlen = sizeof(from);
   b = (recvfrom(s, in, MAX_DGRAM_SIZE, 0, &from, &fromlen));
   if (b == -1) {
      perror("Recvfrom failed");
      return -1;
   }

   printf("Datagram size: %d.\n", b);
   printf("Datagram's IP address is: %s\n", inet_ntoa(from.sin_addr));
   printf("Datagram's port is: %d\n", (int) ntohs(from.sin_port));
   .
   .
   .
}
```

RELATED FUNCTIONS

connect, **getsockopt**, **recv**, **recvmsg**, **send**, **sendmsg**, **sendto**, **setsockopt**

recvmsg Receives Data from a Connected or Unconnected Socket Descriptor

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>

int recvmsg(int s, struct msghdr *mh, int flags);
```

DESCRIPTION

recvmsg receives data from an unconnected or connected socket descriptor **s**. **mh** points to a structure that contains further parameters. The definition of the **msghdr** structure is in the **<sys/socket.h>** header file. The elements of this structure are as follows:

msg_name
points to a structure in which **recvmsg** stores the source address of the message that is being received. This field can be **NULL** if the socket **s** is connected, or if the application doesn't require information on the source address.

msg_namelen
is the length of the buffer pointed to by **msg_name**.

msg_iov
points to an array of **struct iovec** similar to that used by **readv**.

msg_iovlen
is the number of elements in the array pointed to by **msg_iov**.

msg_accrights
is ignored for **AF_INET**.

msg_accrightslen
is ignored for **AF_INET**.

flags consists of the following:

MSG_OOB
processes a single byte of out-of-band data if data are present. If out-of-band data have been moved inline with the **SO_OOBINLINE** socket option, multiple bytes of data can be accessed, and the **MSG_OOB** option is not necessary.

MSG_PEEK
peeks at the incoming message. The data remain available for the next **recvmsg** call.

RETURN VALUE

If **recvmsg** succeeds, it returns the length of the message. If **recv** returns a **0**, it indicates that the connection is closed. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error. It is possible for **recvmsg** to return fewer bytes than the total number specified by the **iovec** elements. This is not an error.

recvmsg Receives Data from a Connected or Unconnected Socket Descriptor
(continued)

CAUTION

recvmsg is an atomic operation. With UDP, no more than one datagram can be read per call. If you are using datagram sockets, make sure that there is enough buffer space in the I/O vector to contain an incoming datagram.

PORTABILITY

mh is commonly documented as **struct msghdr mh[]**, implying that the call operates on an array of these structures. In reality, only one structure is modified by the call. For the purposes of clarity, this technical report documents **mh** in a different way, but the implementation is the same. **recvmsg** is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

connect, **getsockopt**, **recv**, **recvfrom**, **send**, **sendmsg**, **sendto**, **setsockopt**

res_init Initializes the Resolver

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

void res_init(void);
```

DESCRIPTION

res_init initializes the resolver. First, default options are set. Then, a system configuration file is read if one can be located. The configuration file may contain default resolver options for the local site. Refer to Chapter 17, "Network Administration" on page 17-1 for information on default resolver options.

After calling **res_init**, the program may override default site resolver options by accessing the **_res** structure described in Chapter 15, "The BSD UNIX Socket Library" on page 15-1. The program does not need to call **res_init** if **_res** values will not be changed. This routine implements a part of the resolver, which is a set of routines providing a programming interface for communications with Internet name servers.

For information on buffer formats for Internet name servers, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP*.

IMPLEMENTATION

The SAS/C version of **res_init** is a direct port from the BSD UNIX Socket Library.

EXAMPLE

In the following example, **res_init** is used to initialize the resolver.

```
/* Given a hostname as its first parameter, this program prints  */
/* the IP address.                                               */
/* Use of the following resolver options is illustrated:         */
/* The RES_DEBUG option turns on a resolver trace.               */
/* The RES_USEVC option requests use of virtual circuits (TCP)   */
/* when contacting the nameserver.                               */
/* This example assumes that the local site is configured to     */
/* use the resolver (instead of a host file).                    */
/* NOTE: The bitfield(1) compiler option is required.            */
```

res_init Initializes the Resolver
(continued)

```
#include <sys/types.h>
#include <netdb.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <arpa/nameser.h>
#include <resolv.h>
#include <stdio.h>

main(int argc, char *argv[])
{
   struct hostent *hp;
   struct in_addr ip_addr;

      /* Host name should be first parameter.                    */
   if (argc <1 || *argv[1]  == '\0')
      exit(EXIT_FAILURE);

      /* Initialize the resolver, and then set options.          */
   res_init();
   _res.options |= (RES_DEBUG|RES_USEVC);

      /* When gethostbyname uses the resolver, it uses TCP       */
      /* and generates a trace.                                  */
   hp = gethostbyname(argv[1]);
   if (!hp) {
      printf("%s not found\n",argv[1]);
      exit(EXIT_FAILURE);
   }

      /* Host was found.  Print its IP address.                  */
   ip_addr = *(struct in_addr *)(hp->h_addr);
   printf("%s: %s\n",argv[1] ,inet_ntoa(ip_addr));

   exit(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

gethostbyaddr, **gethostbyname**, **gethostent**, **res_send**, **sethostent**

res_mkquery Makes a Domain Name Server Packet

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_mkquery(int op, char *dname, int class, int type, char *data,
                int datalen, struct rrec *newrr, char *buf, int buflen);
```

DESCRIPTION

res_mkquery makes a packet for use with an Internet domain name server and places it in the buffer area pointed to by **buf**, with **buflen** specifying the length in bytes of the buffer. The other seven arguments correspond directly to the fields of a domain name query:

dname
is the domain name.

rrec
is the structure for the passage of new resource records.

class and **type**
are the class and type of the query.

data
gives the address of an array or string of data bytes to be included in the query.

datalen
specifies the integer length in bytes of the data array or string pointed to by **data**.

op is the specified option.
It can be any one of the following query types defined in the **<arpa/nameser.h>** header file:

QUERY	standard query
IQUERY	inverse query
STATUS	name server status query.

This routine implements a part of the resolver, which is a set of routines providing a programming interface for communication with Internet name servers.

For information on buffer formats for Internet name servers, as well as more detailed information about **res_mkquery**, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP, Volume 1*, by Douglas E. Comer, Prentice Hall, 1991.

RETURN VALUE

If successful, **res_mkquery** returns the size of the query. It returns a **-1** if the size of the query is greater than **buflen**.

res_mkquery Makes a Domain Name Server Packet
(continued)

IMPLEMENTATION

The SAS/C version of **res_mkquery** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

dn_comp, **res_init**, **res_send**

res_send Sends a Query

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/nameser.h>
#include <resolv.h>

int res_send(char *msg, int msglen, char *answer, int anslen);
```

DESCRIPTION

res_send sends a query **msg** to name servers and returns an answer. **msglen** is the length of the query. **answer** is an area where the answer to the query can be stored. **anslen** is the length of the answer buffer.

This routine implements a part of the resolver, which is a set of routines providing a programming interface for communications with Internet name servers.

For information on buffer formats for Internet name servers, refer to Chapter 20, "The Domain Name System," in *Internetworking with TCP/IP*.

bitfield(1) option is required for this routine.

RETURN VALUE

If successful, **res_send** returns the size of the answer. It returns a **-1** if there were errors.

IMPLEMENTATION

The SAS/C version of **res_send** is a direct port from the BSD UNIX Socket Library.

RELATED FUNCTIONS

dn_expand, **res_init**, **res_mkquery**

select Determines the Number of Ready Sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, const struct timeval *timeout);
```

DESCRIPTION

select checks to see if any sockets are ready for reading (**readfds**), writing (**writefds**), or have an exceptional condition pending (**exceptfds**). **timeout** specifies the maximum time for **select** to complete. **nfds** specifies the maximum number of sockets to check. If **timeout** is a null pointer, **select** blocks indefinitely. **timeout** points to a **timeval** structure. A **timeout** value of **0** causes **select** to return immediately. This behavior is useful in applications that periodically poll their sockets. The arrival of out-of-band data is the only possible exceptional condition. **fd_set** is a type defined in the **<sys/types.h>** header file. **fd_set** defines a set of file descriptors on which **select** operates. Passing **NULL** for any of the three **fd_set** arguments specifies that **select** should not monitor that condition. On return, each of the input sets is modified to indicate the subset of descriptions that are ready. These may be found by using the **FD_ISSET** macro.

The following macros manipulate sets of socket descriptors:

FD_CLR(fd, &fdset)
removes the socket descriptor **fd** from the socket descriptor set **fdset**.

FD_ISSET(fd, &fdset)
returns nonzero if socket descriptor **fd** is a member of **fdset**. Otherwise, it returns a **0**.

FD_SET(fd, &fdset)
adds socket descriptor **fd** to **fdset**.

FD_SETSIZE
is defined in **<sys/types.h>** as the number of socket descriptors that a process can have open. The default is 64. This value can be increased before you include **<sys/types.h>**. If **FD_SETSIZE** is not increased, the **fd_set** structure will not be defined to be large enough to accommodate socket numbers larger than 63.

FD_ZERO(&fdset)
initializes **fdset** to **0**, representing the empty set.

RETURN VALUE

If **select** succeeds, it returns the number of ready socket descriptors. **select** returns a **0** if the time limit expires before any sockets are selected. If there is an error, **select** returns a **-1**.

select Determines the Number of Ready Sockets
(continued)

CAUTION

The actual timing is not likely to be accurate to the microsecond.

For non-integrated sockets, the SAS/C Library blocks all asynchronous signals during routines that call the TCP/IP communications software. Thus, calls to **select** may leave asynchronous signals blocked for long periods of time.

For integrated sockets, **select** may be interrupted by any unblocked signal, either managed by OpenEdition or SAS/C, in which case **select** will return **-1**, and errno will be set to **EINTR**.

PORTABILITY

select is portable to other environments, including most UNIX systems, that implement BSD sockets.

Unlike UNIX, the SAS/C implementation of **select** does not apply to files, unless you are using integrated sockets. Even if you are using integrated sockets, select has no effect on any non-HFS file descriptors specified. In addition, **select** cannot be used in conjunction with asynchronous I/O, because asynchronous I/O is not supported by the SAS/C Compiler.

EXAMPLE

In this example **select** waits for data to read from a socket.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <stdio.h>

/* This function calls select to wait for data to read from */
/* one of the sockets passed as a parameter.                */
/* If more than 3 seconds elapses, it returns.              */
/* Return value flags. These indicate the readiness of      */
/* each socket for read.                                    */
#define S1READY 0x01
#define S2READY 0X02

waittoread(int s1,int s2)
{
   fd_set fds;
   struct timeval timeout;
   int rc, result;

      /* Set time limit. */
   timeout.tv_sec = 3;
   timeout.tv_usec = 0;
```

select Determines the Number of Ready Sockets
(continued)

```
   /* Create a descriptor set containing our two sockets.  */
   FD_ZERO(&fds);
   FD_SET(s1, &fds);
   FD_SET(s2, &fds);
   rc = select(sizeof(fds)*8, &fds, NULL, NULL, &timeout);
   if (rc==-1) {
      perror("select failed");
      return -1;
   }

   result = 0;
   if (FD_ISSET(s1, &fds)) result |= S1READY;
   if (FD_ISSET(s2, &fds)) result |= S2READY;

   return result;
}
```

RELATED FUNCTIONS

connect, **read**, **recv**, **send**, **write**

selectecb Determines the Number of Ready Sockets

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>
#include <lcsignal.h>

int selectecb(int nfds, fd_set *readfds, fd_set *writefds,
              fd_set *exceptfds, const struct timeval *timeout,
              struct _ecblist *ecblist, size_t ecblcnt)
```

DESCRIPTION

selectecb is identical to **select** with the exception that, in addition to returning when the timeout expires or when activity occurs on one of the specified sockets, **selectecb** also returns if one of the specified ECBs (Event Control Blocks) is posted.

The **ecblist** argument 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 **0**, in which case the ECB array address is ignored.

The declaration for the **_ecblist** structure is as follows:

```
struct _ecblist {
   size_t count;
   unsigned *ecbarr;
}
```

The ECB list is passed via the struct **_ecblist** for several reasons. It allows 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, **_ecblist** facilitates organizing them into arrays; this may slightly improve the performance of **selectecb**, since fewer memory accesses are required to determine the addresses of all the ECBs.

Note that an ECB may be POSTed at any time. Thus, you cannot assume that no ECBs have been POSTed when **selectecb** returns with one or more socket descriptor bits set. An ECB may have been POSTed between the time that **selectecb** was returning and the time that your program checked the return code from **selectecb**.

If **ecblist** is **NULL**, or **ecblcnt** is **0**, or the count field of all the **_ecblist** structures is set to **0**, ECB waiting is not used and the effect is exactly the same as if **select** had been called.

selectecb Determines the Number of Ready Sockets
(continued)

CAUTION

The actual timing is not likely to be accurate to the microsecond. Calls to **selectecb** may leave asynchronous signals blocked for long periods of time.

For non-integrated sockets, the SAS/C Library blocks all asynchronous signals during routines that call the TCP/IP communications software.

For integrated sockets, **selectecb** may be interrupted by any unblocked signal, either managed by OpenEdition or SAS/C, in which case **selectecb** will return **-1**, and errno will be set to **EINTR**. However, execution of **selectecb** is not interrupted by an OpenEdition signal unless the signal terminates the process.

send Sends a Message to a Connected Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int send(int s, const void *buf, int len, int flags);
```

DESCRIPTION

send transmits a message to socket descriptor **s**. The socket must be connected or associated with a foreign peer via the **connect** function. **buf** points to the buffer that contains the message. **len** is the number of bytes to be sent.

flags consists of the following:

MSG_OOB
requests that message buffers be sent as out-of-band data.

MSG_DONTROUTE
bypasses routing; uses the network portion of the destination address to select a network interface.

RETURN VALUE

If **send** succeeds, it returns the number of characters sent. If there is an error, it returns a **-1**.

PORTABILITY

send is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

recv, **recvfrom**, **recvmsg**, **sendmsg**, **sendto**, **write**

sendmsg Sends a Message to a Connected or Unconnected Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int sendmsg(int s, struct msghdr *mh, int flags)
```

DESCRIPTION

sendmsg sends a message to an unconnected or connected socket **s**. **mh** points to a structure containing further parameters. The definition of the **msghdr** structure is in the **<sys/socket.h>** header file. The elements of this structure are as follows:

msg_name
points to a structure in which **sendmsg** stores the source address of the message that is being received. This field can be **NULL** if the socket **s** is connected, or if the application doesn't require information on the source address.

msg_namelen
is the length of the buffer pointed to by **msg_name**.

msg_iov
points to an array of **struct iovec** similar to that used by **readv**.

msg_iovlen
is the number of elements in the array pointed to by **msg_iov**.

msg_accrights
is ignored for **AF_INET**.

msg_accrightslen
is ignored for **AF_INET**.

flags consists of the following:

MSG_OOB
requests that message buffers be sent as out-of-band data.

MSG_DONTROUTE
bypasses routing; uses the network portion of the destination address to select a network interface.

RETURN VALUE

If **sendmsg** succeeds, it returns the length of the message. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

CAUTION

sendmsg is an atomic operation. With UDP, no more than one datagram can be read per call. If you are using datagram sockets, make sure that there is enough buffer space in the I/O vector to contain an incoming datagram.

sendmsg Sends a Message to a Connected or Unconnected Socket
(continued)

PORTABILITY

mh is commonly documented as **struct msghdr mh[]** , implying that the call operates on an array of these structures. In reality, only one structure is modified by the call. For the purposes of clarity, this manual documents **mh** in a different way, but the implementation is the same. **sendmsg** is portable to other environments, including most UNIX systems, that implement BSD sockets.

EXAMPLE

In this example, **sendmsg** is used to transmit banking transactions.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>

#include "transdef.h"                    /* application header file  */

/* This routine writes banking transactions to one of several         */
/* regional servers (chosen based on the contents of the transaction, */
/* not based on the location of the local host). "s" is a datagram    */
/* socket. "head" and "trail" are application level transaction       */
/* header and trailer components. "trans" is the body of the          */
/* transaction. Reliability must be ensured by the caller (because    */
/* datagrams do not provide it). The server receives all three        */
/* parts as a single datagram.                                        */
puttrans(int s, struct header *head, struct record *trans,
         struct trailer *trail)
{
   int rc;

      /* socket address for server                                    */
   struct sockaddr_in dest;

      /* will contain information about the remote host               */
   struct hostent *host;
   char fullname[64];

      /* Will point to the segments of the (noncontiguous)            */
      /* outgoing message.                                            */
   struct iovec iov[3];

      /* This structure contains parameter information for sendmsg.   */
   struct msghdr mh;
```

sendmsg Sends a Message to a Connected or Unconnected Socket
(continued)

```
   /* Choose destination host from region field of the data       */
   /* header. Then find its IP address.                           */
strcpy(fullname,head->region);
strcat(fullname,".mis.bank1.com");
host = gethostbyname(fullname);
if (host==NULL) {
    printf("Host %s is unknown.\n",fullname);
    return -1;
}

/* Fill in socket address for the server. We assume a       */
/* standard port is used.                                                  */
memset(&dest,'\0',sizeof(dest));
dest.sin_family = AF_INET;
memcpy(&dest.sin_addr,host->h_addr,sizeof(dest.sin_addr));
dest.sin_port = htons(TRANSACTION_SERVER);

   /* Specify the components of the message in an "iovec".   */
iov[0] .iov_base = (caddr_t)head;
iov[0] .iov_len = sizeof(struct header);
iov[1] .iov_base = (caddr_t)trans;
iov[1] .iov_len = sizeof(struct record);
iov[2] .iov_base = (caddr_t)trail;
iov[2] .iov_len = sizeof(struct trailer);

   /* The message header contains parameters for sendmsg.    */
mh.msg_name = (caddr_t) &dest;
mh.msg_namelen = sizeof(dest);
mh.msg_iov = iov;
mh.msg_iovlen = 3;
mh.msg_accrights = NULL;           /* irrelevant to AF_INET */
mh.msg_accrightslen = 0;           /* irrelevant to AF_INET */

rc = sendmsg(s, &mh, 0);           /* no flags used        */
if (rc == -1) {
   perror("sendmsg failed");
   return -1;
}
return 0;
}
```

RELATED FUNCTIONS

connect, **getsockopt**, **ioctl**, **recv**, **recvfrom**, **recvmsg**, **send**, **sendto**, **setsockopt**

sendto Sends a Message to a Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int sendto(int s, const void *buf, int len, int flags,
           const void *to, int addrlen);
```

DESCRIPTION

sendto transmits a message from the area pointed to by **buf** of length **len** to socket descriptor **s**. **to** is the target address; **addrlen** is the address size. **sendto** can be used on any type of socket. It is most commonly used for unconnected datagram sockets for which different destinations may be associated with different datagrams.

flags consists of the following:

MSG_OOB
requests that message buffers be sent as out-of-band data.

MSG_DONTROUTE
bypasses routing; uses the network portion of the destination address to select a network interface.

RETURN VALUE

If **sendto** succeeds, it returns the number of characters sent. If there is an error, it returns a **-1**.

PORTABILITY

sendto is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

recv, **recvfrom**, **recvmsg**, **send**, **sendmsg**, **write**

sethostent Opens a Host File or Connects to a Name Server

SYNOPSIS

```
#include <netdb.h>

void sethostent(int stayopen);
```

DESCRIPTION

sethostent opens a host file or begins a connection with the name server.

In some instances, when the resolver is used to look up the host name, a virtual circuit (that is, a TCP connection) is used to communicate with the name server. This is based on the **RESOLVEVIA** statement in the **TCPIP.DATA** file or the **RES_USEVC** resolver option specified by your program. In these cases, you can use the **RES_STAYOPEN** resolver option to maintain the connection with the name server between queries.

In other instances, the host file is used as a source of host names. In this case, **sethostent** rewinds the file. This enables successive **sethostent** calls to read the host file sequentially. Specifying the **stayopen** parameter with **sethostent** causes the host file to remain open between **gethostbyname** and **gethostbyaddr** calls.

Refer to Chapter 17, “Network Administration” on page 17-1 for information on naming host files and the logic that determines whether the host file or the resolver is used for looking up names.

RETURN VALUE

sethostent does not return a value.

PORTABILITY

The logic that determines whether to use the host file or the resolver is not uniform across environments. At the source code level, however, **sethostent** is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

sethostent is a combination of the host file and resolver versions of the BSD UNIX Socket Library **sethostent**.

RELATED FUNCTIONS

endhostent, **gethostent**, **gethostbyname**

setnetent Opens the Network File

SYNOPSIS

```
#include <netdb.h>

void setnetent(int stayopen);
```

DESCRIPTION

setnetent opens the network file, that is, a file with the same format as **/etc/networks** in the UNIX environment. If the file is already open, **setnetent** rewinds the file so that succeeding **getnetent** calls can read it sequentially. Specifying a nonzero value for **stayopen** causes the network file to remain open between subsequent **getnetbyname** and **getnetbyaddr** calls. Refer to Chapter 17, "Network Administration" on page 17-1 for information on naming the network file for your system.

RETURN VALUE

setnetent does not return a value.

PORTABILITY

setnetent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

This routine is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getnetent, **endnetent**, **getnetbyname**, **getnetbyaddr**

setprotoent Opens the Protocols File

SYNOPSIS

```
#include <netdb.h>

void setprotoent(int stayopen);
```

DESCRIPTION

setprotoent opens the protocols file, that is, a file with the same format as **/etc/protocols** in the UNIX environment. If the file is already open, **setprotoent** rewinds the file so that succeeding **getprotoent** calls can read it sequentially. Specifying a nonzero value for **stayopen** causes the protocols file to remain open between subsequent **getprotobyname** and **getprotobynumber** calls. Refer to Chapter 17, "Network Administration" on page 17-1 for information on naming the protocols file for your system.

RETURN VALUE

setprotoent does not return a value.

PORTABILITY

setprotoent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

This routine is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

endprotoent, **getprotobyname**, **getprotobynumber**, **getprotoent**

setrpcent Opens the /etc/rpc Protocols File

SYNOPSIS

```
#include <netdb.h>

int setrpcent(int stayopen);
```

DESCRIPTION

setrpcent opens the Sun RPC program numbers file, that is, a file with the same format as **/etc/rpc** in the UNIX environment. If the file is already open, **setrpcent** rewinds the file so that succeeding **getrpcent** calls can read it sequentially. Specifying a nonzero value for **stayopen** causes the protocols file to remain open between subsequent **getrpcbyname** and **getrpcbynumber** calls. Refer to "/etc/rpc" on page 17-3 and Chapter 17, "Network Administration" on page 17-1 for information on naming this file for your system.

RETURN VALUE

setrpcent does not return a value.

CAUTION

The value that **getrpcbynumber** returns points to a static structure within the library. You must copy the information from this structure before you make further **getrpcbyname**, **getrpcbynumber**, or **getrpcent** calls.

PORTABILITY

setrpcent is portable to other systems that support Sun RPC 4.0.

IMPLEMENTATION

This function is built from the Sun RPC 4.0 distribution.

RELATED FUNCTIONS

endrpcent, **getrpcbyname**, **getrpcbynumber**, **getrpcent**

setservent Opens the Services File

SYNOPSIS

```
#include <netdb.h>

void setservent(int stayopen);
```

DESCRIPTION

setservent opens the services file, that is, a file with the same format as **/etc/services** in the UNIX environment. If the file is already open, **setservent** rewinds the file so that succeeding **getservent** calls can read it sequentially. Specifying a nonzero value for **stayopen** causes the protocols file to remain open between subsequent **getservbyname** and **getservbyaddr** calls. Refer to Chapter 17, "Network Administration" on page 17-1 for information on naming the services file for your system.

RETURN VALUE

setservent does not return a value.

PORTABILITY

setservent is portable to other environments, including most UNIX systems, that implement BSD sockets.

IMPLEMENTATION

This routine is ported directly from the BSD UNIX Socket Library.

RELATED FUNCTIONS

getservent, **endservent**, **getservbyname**, **getservbyport**

setsockimp Specifies the Socket Implementation

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int setsockimp(const char *name);
```

DESCRIPTION

setsockimp defines the name of the socket implementation. It must be called before calling any other socket function. The argument **name** is a one-to-four character string identifying the socket implementation. If name is **''COM''**, the standard non-integrated socket implementation at your site is used. If name is **''OE''**, the OpenEdition integrated socket implementation is used. Your TCP/IP vendor may define other names to access a specific implementation. The name string is used to construct the name of a load module (LSCN*name*) implementing the socket interface.

If **setsockimp** is not called before socket functions are used, a default socket implementation is used. For programs invoked with **exec**-linkage, the default implementation is OpenEdition integrated sockets. For other programs, the default is non-integrated sockets.

RETURN VALUE

setsockimp returns **0** if successful or **-1** if unsuccessful. If **setsockimp** fails because an invalid implementation name was specified, socket functions can still be called, and the default implementation will be used.

setsockopt Sets Socket Options

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int setsockopt(int s, int level, int optname, const void *optval,
int optlen);
```

DESCRIPTION

setsockopt sets the value of the options associated with socket descriptor **s**. **level** is the level of the option. **optname** is the name of the option. **optval** is a pointer to a buffer that receives the value of the requested option. As an input parameter, the integer **optlen** should be set to the size of the **optval** buffer.

Most options are at the socket level. Pass **SOL_SOCKET** as the **level** parameter for these options:

SO_LINGER
sets the **linger** option. The **linger** option controls the behavior of the **close** call when there are some data that have not been sent. **optval** should point to a **struct linger**. If the value of the **l_onoff** field is **0**, **close** returns immediately. In this case, TCP/IP continues to attempt to deliver the data, but the program is not informed if delivery fails. If the value of the **l_onoff** field is nonzero, the behavior of the **close** call depends on the value of the **l_linger** field. If this value is **0**, unsent data are discarded at the time of the **close**. If the value of **l_linger** is not **0**, the **close** call blocks the caller until there is a timeout or until all data are sent. The **l_linger** field indicates the length of the desired timeout period. Some TCP/IP implementations may ignore the value of the **l_linger** field. **s** must be a stream socket.

SO_OOBINLINE
receives out-of-band data inline, that is, without setting the **MSG_OOB** flag in **recv** calls. **optval** should point to an integer. **s** must be a stream socket. Refer to "Out-of-Band Data" in Chapter 6, "Berkeley Sockets," in *UNIX Network Programming* for information on out-of-band data.

SO_REUSEADDR
allows local port address to be reused. **optval** should point to an integer. **s** must be a stream socket.

The option level **IPPROTO_TCP** is available for one option. The **TCP_NODELAY** option indicates that TCP's normal socket buffering should not be used on the socket. The **TCP_NODELAY** option is not operative; it is supported for source code compatibility. The **<netinet/in.h>** and **<netinet/tcp.h>** headers files are required for this option.

RETURN VALUE

If **setsockopt** succeeds, it returns a **0**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

setsockopt Sets Socket Options
(continued)

PORTABILITY

setsockopt is portable to other environments, including most UNIX systems, that implement BSD sockets. The supported options may vary depending on the system; additional options may be supported by a specific TCP/IP software product, and the options described above may be supported differently. Refer to your software documentation for details.

RELATED FUNCTIONS

bind, **close**, **ioctl**, **recv**, **socket**

shutdown Ends Communication with a Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int shutdown(int s, int how);
```

DESCRIPTION

shutdown ends communication on socket descriptor **s** in one or both directions. **how** specifies the shutdown condition and can be specified in the following ways:

0
does not allow any more receives.

1
does not allow any more sends.

2
does not allow any more sends or receives.

RETURN VALUE

If **shutdown** is successful, it returns a **0**; otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

shutdown is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

close

socket Creates a Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

DESCRIPTION

socket creates an endpoint for communication and returns a descriptor, which is a small integer used to reference the socket.

domain
specifies the communications protocol family, using a symbolic name defined in the **<sys/socket.h>** header file. The defined families are as follows:

AF_INET
specifies ARPA Internet protocols.

AF_UNIX
specifies UNIX domain protocols. (Only used with integrated sockets.)

type
specifies the semantics of communication, as defined in **<sys/socket.h>**. The defined types are as follows:

SOCK_STREAM
provides sequenced, reliable byte streams based on two-way connections. This type also supports out-of-band transmission mechanisms.

SOCK_DGRAM
supports datagrams, which are connectionless messages of a fixed maximum length. The delivery of datagrams is unreliable. The application is responsible for acknowledgment and sequencing.

SOCK_RAW
provides access to low-level network protocols and interfaces.

A stream socket must be connected to another socket with a **connect** call before any data can be sent or received on it. Data are transferred using **read** or **write** calls or variants of the **send** and **receive** calls. The socket can be closed with a **close** call when the session is completed, or the socket will be closed automatically at program exit.

protocol
specifies the protocol to be used. If **protocol** is set to **0**, the system selects the default protocol number for the socket domain and type specified. This is usually adequate, since the most common socket types support only one protocol. **getprotobyname** and related calls can also be used to select the protocol.

RETURN VALUE

If **socket** is successful, it returns a descriptor referencing the socket; otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

socket Creates a Socket
(continued)

PORTABILITY

socket is portable to other environments, including most UNIX systems, that implement BSD sockets.

EXAMPLE

The following is an example of a complete socket program.

```
/* This is an example of a complete socket program. It is a      */
/* client program for the finger server which runs on many UNIX  */
/* systems. This client program returns information about logged */
/* on users for any remote system that is running the finger     */
/* server. The format of the command is:                         */          */
/*                                                               */
/*        finger [-l] hostname                                   */
/*                                                               */
/* The output is typically identical to that which one would     */
/* receive when running the "finger" command locally on that     */
/* system. Specification of the -l option results in more        */
/* verbose output.                                               */

#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>

   /* No need to translate characters on ASCII systems.          */
#ifndef  __SASC__
#define ntohcs(c) (c)
#define htoncs(c) (c)
#endif

   /* Define some useful constants.                              */
#define FALSE 0
#define TRUE 1
#define READ_BLOCK_SIZE 256
#define OUTBUF_SIZE 2
#define FINGER_PORT 79
#define ASCII_CR 0x0D
#define ASCII_LF 0x0A
```

socket Creates a Socket
(continued)

```
main(int argc, char *argv[])
{
   int n,i;

   /* Socket descriptor for communication with the finger server.*/
   int s;

   /* Socket address for server.                                  */
   struct sockaddr_in sa;

     /* Use to send ASCII CR and LF characters required by       */
     /* the protocol.                                            */
   char cr = ASCII_CR, lf = ASCII_LF;

     /* buffers  */
   char in[READ_BLOCK_SIZE] ;
   char out[OUTBUF_SIZE] ;

     /* option status  */
   int longlist = FALSE;
   char *op;

     /* will contain information about the finger service        */
   struct servent *serv;

     /* will contain information about the remote host           */
   struct hostent *host;

   if (*++argv && *argv[0] == '-') {     /* argument processing */
      op = *argv;
      if (op[1] == 'l' || op[1] == 'L')
         longlist = 1;
      else
          printf("finger: unsupported option \ "%c\"",op[1])
      argv++;
   }
   if (!*argv) {
      printf("finger: hostname missing");
      exit(EXIT_FAILURE);
   }

      /* Assume *argv now points to hostname.                     */
      /* Find the IP address for the requested host.              */
   host = gethostbyname(*argv);
   if (host==NULL) {
       printf("finger: Host %s is unknown.\n",*argv );
       exit(EXIT_FAILURE);
   }
```

socket Creates a Socket
(continued)

```
   /* Find the port for the finger service. If the services file */
   /* is not available, we will use the standard number. We      */
   /* specify "tcp" as the protocol. This call attempts to       */
   /* find the services file.                                    */
serv = getservbyname("finger", "tcp");

   /* Prepare a socket address for the server. We need an IP     */
   /* address(corresponding to the given host name) and a port   */
   /* number (corresponding to the "finger" service).         */
memset(&sa,'\0',sizeof(sa));
sa.sin_family = AF_INET;
memcpy(&sa.sin_addr,host->h_addr,sizeof(sa.sin_addr));
sa.sin_port = serv ? serv->s_port : htons(FINGER_PORT);

   /* Get a socket of the proper type. We use SOCK_STREAM     */
   /* because we want to communicate using TCP (we want  a    */
   /* reliable protocol).                                                  */
s = socket(AF_INET, SOCK_STREAM, 0);
if (s == -1) {
   perror("finger - socket() failed");
   exit(EXIT_FAILURE);
}

   /* Connect to the host and port.                           */
if (connect(s, &sa, sizeof(sa)) == -1) {
   perror("finger - connect() failed");
   return(EXIT_FAILURE);
}

   /* read() and write() are the most convenient calls  for   */
   /* transmitting and receiving data over stream sockets.    */

   /* Write to server.                                        */
   /* If long option is specified, pass that first.           */
if (longlist) {
   out[0]  = htoncs('/');
   out[1]  = htoncs('W');
   write(s,out,2);
}
   /* Terminate msg with CR-LF. */
write(s,&cr,1);
write(s,&lf,1);
```

socket Creates a Socket
(continued)

```
      /* Server should respond.    */
      /* Read until EOF indicated. */
   while (n=read(s,in,READ_BLOCK_SIZE)) {
      for (i=0; i<n ; i++) {
         if (in[i] ==ASCII_CR) continue;
            /* Presume part of CR-LF pair.                    */
         putchar(ntohcs(in[i] ));
      }
   }

      /* Close the socket.         */
   close(s);

}
```

RELATED FUNCTIONS

accept, **bind**, **close connect**, **getsockname**, **getsockopt**, **givesocket**, **listen**, **open**, **read**, **recv**, **recvfrom**, **recvmsg**, **select**, **send**, **sendmsg**, **sendto**, **setsockopt**, **shutdown**, **takesocket**, **write**, **writev**

socketpair Creates a Connected Socket Pair

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int fd[2]);
```

DESCRIPTION

socketpair creates a connected pair of unnamed sockets. The arguments are:

domain
specifies the addressing family that will be used to interpret addresses. **AF_UNIX** must be specified.

type
specifies the communication service type: either **SOCK_STREAM** or **SOCK_DGRAM** may be specified.

protocol
specifies the protocol to be used with the socket pair. This number corresponds to the **p_proto** field of the **protoent** structure. If a **0** is specified, the system will select the protocol. Refer to Chapter 15, "The BSD UNIX Socket Library" on page 15-1 for information about the **protoent** structure.

fd
is an array used to hold the file descriptors for the sockets.

RETURN VALUES

If **socketpair** is successful, it returns **0**; otherwise, it returns a **−1** and sets **errno** to indicate the type of error.

CAUTION

socketpair is only vaild when integrated sockets are in use.

PORTABILITY

socketpair is portable to other environments, including most UNIX systems that implement BSD sockets.

socktrm Terminates the Socket Library Environment

SYNOPSIS

```
#include <sys/socket.h>

void socktrm(void)
```

DESCRIPTION

socktrm terminates the current SAS/C Socket Library environment and returns it to an uninitialized state without requiring termination of the general SAS/C Library environment.

IMPLEMENTATION

For IBM TCP/IP nonintegrated sockets that rely upon IUCV message passing, the IUCV path to IBM TCP/IP is severed and cleared, which results in termination of all socket connections held by the current environment and disestablishes the IUCV connection with IBM TCP/IP.

For integrated sockets, calling **socktrm** has no effect since socket status is controlled by OpenEdition rather than by the SAS/C Library. Any open sockets remain open.

For CICS sockets, the effect of calling **socktrm** is to close all open socket connections and mark the socket library state as uninitialized. For CICS, since no IBM TCP/IP CICS call is provided to directly undo the libraries', internal CICS socket initialization INITAPI call, further socket calls by the application may result in errors indicating that a second INITAPI was attempted with a duplicate name. Note that the current IBM CICS TCP/IP behavior is to free up the name for reuse after a duplicate INITAPI is attempted; however, the library does not take advantage of this behavior.

CAUTIONS

All outstanding socket connections are closed and lost upon completion of this function call.

PORTABILITY

socktrm is specific to the SAS/C sockets implementation for use with IBM TCP/IP and, therefore, is not portable. Other TCP/IP vendors and socket implementations, at their discretion, may provide an equivalent function call or different functionality.

RELATED FUNCTIONS

close, **shutdown**

takesocket Takes a Socket Descriptor from Donor Process

SYNOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int takesocket(struct clientid *clientid, int s);
```

DESCRIPTION

takesocket takes a socket descriptor, **s**, and a pointer to a client structure, **clientid**, and returns a local client socket descriptor, which must be used for subsequent TCP/IP calls. **s** must be a connected stream socket. The donor process must have already obtained client ID for the receiving process, and must have already called **givesocket**. The mechanism for exchanging client IDs and the socket descriptor number is the responsibility of the two processes and is not accomplished by either the **givesocket** or the **takesocket** call.

When giving a socket, the donor may use a **clientid** structure that contains blanks as the subtask name. However, when taking a socket, the receiver must specify the subtask name of the donor in the client ID. This allows the donor to make the socket available to any number of subtasks while the receiver must have the actual subtask name to take the socket. The subtask name effectively acts as a key.

The donor process should not close the socket before the receiving process has completed the **takesocket** call. The receiving process must have a means of informing the donor that the **takesocket** call has completed. Using **select** or **selectecb** in the donor, frees the receiver of this responsiblility.

Note: **takesocket** is only supported with non-integrated sockets.

RETURN VALUE

If **takesocket** is successful, it returns a non-negative, client socket descriptor, which can be different from the donor descriptor, **s**. Otherwise, it returns a **-1**, and sets **errno** to indicate the type of error.

PORTABILITY

takesocket is not portable to UNIX operating systems. On UNIX operating systems, sockets are typically transferred from parent to child processes as a side effect of the **fork()** system call.

EXAMPLE

Refer to the **getclientid** function for an example that illustrates the use of **takesocket**.

RELATED FUNCTIONS

getclientid, **givesocket**

writev Writes Data from an Array of Buffers to a Socket

SYNOPSIS

```
#include <sys/types.h>
#include <sys/uio.h>
#include <fcntl.h>

int writev(int s, struct iovec *iov, int iovcnt);
```

DESCRIPTION

writev writes data to connected socket descriptor **s** from the **iovcnt** buffers specified by the **iov** array. As with the **write** call, the socket must have been previously associated with a remote address via the **connect** system call. If there are no data, **writev** blocks the caller unless the socket is in non-blocking mode. The **iovec** structure is defined in the **<sys/uio.h>** header file. Each **iovec** entry specifies the base address and length of an area in memory from which the data should be taken. **writev** completely sends one area before proceeding to the next area.

writev is an atomic operation. For datagram sockets, each **writev** call causes one datagram to be sent.

Note: Although **writev** is primarily used with sockets, it can also be used to write any file that can be accessed by the **write** function.

RETURN VALUE

If **writev** succeeds, it returns the number of bytes written. If **writev** returns a **0**, the end-of-file has been reached. If **writev** fails, it returns a **-1**.

PORTABILITY

writev is portable to other environments, including most UNIX systems, that implement BSD sockets.

RELATED FUNCTIONS

connect, **recv**, **recvfrom**, **recvmsg**, **read**, **readv**, **write**

Part 3

SAS/C® POSIX Support

Chapters

19 Introduction to POSIX
20 POSIX Function Reference
Function Index

19 Introduction to POSIX

19-1 POSIX and OpenEdition Concepts
- *19-1 Process*
- *19-1 Permissions*
- *19-2 Signals*
- *19-2 Files*
- *19-2 Directories*
- *19-3 Links*
- *19-3 Terminals and Sessions*
- *19-3 Shells*

19-4 The errno Variable
- *19-4 POSIX and OpenEdition Error Numbers*
- *19-5 Internal Error Numbers*

19-6 SAS/C OpenEdition Interfaces
- *19-11 MVS Considerations*
- *19-12 fork*
- *19-12 exec*
- *19-14 Standard Types*
- *19-14 HFS Files and DDnames*
- *19-14 Signal Handling and ABENDs*
- *19-15 Multiple Processes in an Address Space*
- *19-15 Behavior when OpenEdition is not Available*

POSIX and OpenEdition Concepts

The POSIX 1003.1 standard is an ISO standard that specifies operating system functionality in a C language interface. With IBM's 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. References in this book to POSIX are to the ISO/ANSI C standards or draft standards and not to a particular implementation.

POSIX 1003.1 is based on a UNIX operating system standard. A discussion of each of the fundamental POSIX concepts follows.

Process

A *process* is an abstraction that represents an executing program. Multiple processes execute independently and have separate address spaces. Processes can create, interrupt, and terminate other processes, subject to security restrictions. A process has characteristics in common with both an MVS address space and an MVS task. It is easier for processes to create or influence each other than for an MVS address space to create or influence another address space.

Note: MVS 5.1 allows more than one process to be created in the same address space. This is an extension of the POSIX process model.

Permissions

Each user of a POSIX system has a defined user ID and group ID. User IDs and group IDs are integers; user names and group names are strings. Permissions are defined in terms of the user ID and group ID. For example, the permissions for a file can be defined as readable and writable by the owner, by other members of the owning group, or by anyone.

Programs can have permissions that are associated with the ownership of the program rather than with the user running the program. The real user ID and real group ID are associated with the user running the program. The effective user ID and effective group ID, which could be affected by the program, are used to determine the current permissions for the process. The POSIX permission structure is coarser than the structure defined by mainframe security products such as RACF; the ability to create a program with the privileges of its owner is new to MVS. POSIX allows users with appropriate privileges to suppress certain permissions checks; users with such privileges are called *super-users.*

Signals

A *signal* is an interruption of a process. Signals can be generated by the operating system; they may also be generated by one process and sent to another. In general, a process can catch a signal to take some program-specified action. If a program does not catch a signal, the system takes some default action, which is most frequently to terminate the process; the default action can also be to do nothing, or either to halt or resume process execution.

Some signals cannot be caught. The signal **`SIGKILL`** can be used to terminate another process without allowing the target process to intercept termination. Unless the user has appropriate privileges, a process can send a signal only to other closely related processes, for instance, ones that it created. MVS does not have a concept that corresponds to the POSIX signal. Under MVS, various types of interrupts are handled by a number of unrelated facilities such as **`ESTAE`**, **`STIMER`**, and **`STAX`**.

Files

A POSIX *file* is a stream of bytes. A *regular file* is stored on disk and supports random access. A *special file* has special properties. An example of a special file is a user's terminal. Another example is a *pipe*, which is a software connection between two processes; characters written to the pipe by one process can be read by the other process.

Processes can share files. A locking mechanism allows processes to cooperate in sharing a file. Filenames may be as long as 1024 bytes in OpenEdition. Case distinctions are honored in filenames; **`calc.c`** and **`Calc.c`** reference separate files. This is in contrast to MVS data sets, which are record-oriented, cannot generally be shared, and whose names are limited to a small number of uppercase characters. POSIX files are organized into file systems that are stored in MVS data sets by OpenEdition. These data sets can be mounted independently.

Directories

POSIX files are organized into directories. A *directory* is a file that contains a list of other files and their attributes. Directories can reference subdirectories to any number of levels. A complete pathname includes a directory specification and a filename. The pathname **`/u/fred/calc.c`** refers to the file named **`calc.c`**, in the directory **`fred`**, in the directory **`u`** in the system root directory. Both special and regular files are referenced by hierarchical filenames. For example, the pathname for the user's terminal is **`/dev/tty`**. Each process has a current working directory that defines a reference point for pathnames. If a working directory for a process is **`/u/fred`**, the pathname **`calc.c`** is interpreted as **`/u/fred/calc.c`**. You can read a directory and write applications that process some or all of the files in a directory. Neither MVS catalogs nor PDS directories offer the flexibility and convenience of POSIX directories.

Links

POSIX permits more than one pathname to refer to the same file. These additional pathnames are called *links.* Links to a file need not reside in the same directory as the file. A file continues to exist as long as there is a link to it. A related MVS concept is alias data set names, but, because of the differences between catalogs and directories, the correspondence is not very close. The POSIX 1003.1a draft standard introduces the symbolic link concept. A *symbolic link* is a link based on the name of the file that is linked to. Unlike regular links, it is possible to have a symbolic link to a file in a different file system from the link.

Terminals and Sessions

A POSIX terminal is a special file. POSIX provides interfaces that are valid only for terminal files. For example, **`tcflush`** is an interface that is used to discard unread terminal input. A *terminal* may be either a physical ASCII asynchronous terminal, such as a VT-100, or a pseudo-terminal, which is a software emulation of an asynchronous terminal. OpenEdition supports only pseudo-terminals created by the TSO OMVS command. When a process writes to an OpenEdition pseudo-terminal, the output is received by the OMVS command, which then writes the data to the user's TSO terminal. Input is handled in the same way. Some POSIX terminal control functions, such as defining the terminal's baud rate, have no effect on OpenEdition because they are not meaningful for a pseudo-terminal.

A *controlling terminal* is a POSIX terminal that controls a set of processes called a *session.* In a normal UNIX system, a session is started when a user logs in; the controlling terminal is the terminal associated with the login. In OpenEdition, a session is usually started when a user executes the OMVS command.

A session may be divided into several process groups: a foreground process group and one or more background process groups. Processes are generally allowed to send signals only to other processes in their process group. The controlling terminal can use special input sequences to interrupt or halt the processes of the foreground process group. For example, the control-C character causes a SIGINT to be sent to each foreground process. Note that the OMVS command uses the sequence **`¢c`** to replace control-C, which cannot be entered from a 3270 keyboard.

Shells

The *shell* is a specialized application that is used to invoke other programs; it implements a scripting language that can be used in a similar fashion to CLIST or REXX under TSO. The shell is a UNIX construct associated with OpenEdition. It is defined by a related POSIX standard, 1003.2. Although the shell is not defined by the POSIX 1003.1 standard, it uses many interfaces that are defined by that standard. Under MVS, invoking an application from the shell is the best way to run an application in a fully POSIX-compliant environment. The shell resembles the TSO TMP (terminal monitor program). Because the shell is not part of POSIX 1003.1 functionality, the SAS/C Library does not interact directly with it, with the exception of the **`system`** and **`popen`** functions, which can invoke an OpenEdition command through the shell.

The errno Variable

As described in in Chapter 1, "Introduction to the SAS/C Library," in *SAS/C Library Reference, Volume 1*, 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>`**.

POSIX and OpenEdition Error Numbers

The **<errno.h>** header file contains definitions for the macro identifiers that name system error status conditions. When a SAS/C Library function sets an error status by assigning a nonzero valued to **errno**, the calling program can check for a particular value by using the name defined in **<errno.h>**.

The following OpenEdition related, POSIX defined **errno** values are defined in **<errno.h>**:

E2BIG	argument list for **exec** function too large
EAGAIN	resource temporarily unavailable
EBUSY	resource busy (synonym for **EINUSE**)
ECHILD	child process not found
EDEADLK	resource deadlock avoided
EFAULT	invalid argument address
EFBIG	file too large
EINVAL	invalid function argument (synonym for **EARE**)
EISDIR	output file is a directory
ELOOP	too many symbolic links in pathname (1003.1a)
EMFILE	open file limit exceeded (synonym for **ELIMIT**)
EMLINK	system limit on links exceeded
EMVSEXPIRE	expired password (non-POSIX extension)
EMVSINITIAL	error in establishing OpenEdition process (non-POSIX extension)
EMVSNOTUP	OpenEdition kernel is not active (non-POSIX extension)
EMVSPASSWORD	incorrect password (non-POSIX extension)
ENAMETOOLONG	filename too long
ENDENT	file or directory not found (synonym for **ENFOUND**)
ENFILE	too many open files in system
ENODEV	inappropriate use of device
ENOEXEC	attempt to execute non-executable file
ENOLCK	no HFS record locks were available
ENOSYS	function not implemented by system
ENOTDIR	pathname component not a directory
ENOTEMPTY	directory not empty
ENOTTY	file is not a terminal
ENXIO	non-existent or inappropriate device
EPERM	operation not permitted
EPIPE	write to pipe without headers
EROFS	file system mounted read only
ESPIPE	seek to unseekable file (synonym for **EUNSUPP**)
ESRCH	process not found
EXDEV	link from one file system to another
GIO	input/output error (synonym for **EDEVICE**)

Note: In addition to the values listed above, common SAS/C **errno** values are listed in Chapter 1, "Introduction to the SAS/C Library," in *SAS/C Library Reference, Volume 1.* Also refer to "<errno.h>" on page 15-4 for socket-related **errno** values. For a complete listing of all **errno** values, see the *SAS/C Compiler and Library Quick Reference Guide.*

Also note that OpenEdition frequently uses a reason code to provide additional information about errors occuring in POSIX functions. See "System Macro Information" in Chapter 1 of *SAS/C Library Reference, Volume 1* for information on accessing OpenEdition reason codes.

Internal Error Numbers

The following **errno** values represent OpenEdition internal errors. Contact IBM Technical Support if you receive one of these errors. These **errno** values are described in more detail in the manual, *OpenEdition Assembler Callable Services.*

EIBMCANCELLED

EIBMCONFLICT

EIBMSOCKINUSE

EIBMSOCKOUTOFRANGE

EMVSCATLG

EMVSCVAF

EMVSDYNALC

EMVSPFSFILE

EMVSPFSPERM

EMVSSAF2ERR

EMVSSAFEXTRERR

EOFFLOADBOXDOWN

EOFFLOADBOXERROR

EOFFLOADBOXRESTART

Consult *The POSIX.1 Standard: A Programmer's Guide* by Fred Zlotnick (1991, Benjamin/Cummings Publishing) or the POSIX standards for information on the POSIX **errno** values; consult IBM OpenEdition documentation for information on the EMVS- and EIBM- **errno** values.

Because the **errno** values resulting from POSIX functions are well organized and well-defined, the SAS/C Library does not diagnose many problems that can occur with these functions, even when it diagnoses similar problems occurring with MVS files. This eases porting programs from other open systems, since these systems do not, in general, have library diagnostics. In general, the SAS/C Library will issue diagnostics for failures of OpenEdition functionality only when (a) the failure is certainly the result of programmer error, or (b) the error is so esoteric that diagnosis is unlikely without a message.

SAS/C OpenEdition Interfaces

SAS/C functions take advantage of OpenEdition functionality in the following categories:

process control functions	create and manipulate POSIX processes.
permission functions	query or manipulate POSIX permissions.
signal-handling functions	manipulate POSIX and non-POSIX signals; they are described in *SAS/C Library Reference, Volume 1.*
I/O functions	perform I/O to POSIX files. ANSI I/O functions can also be used to access POSIX files; they are described in *SAS/C Library Reference, Volume 1.*
directory functions	manipulate POSIX directories.
file utilities	perform non-I/O functions on POSIX files. These functions are described in *SAS/C Library Reference, Volume 1.*
terminal functions	query or manipulate terminal files.
sessions and process groups functions	query or manipulate process groups and sessions.
environmental interfaces	are interfaces to unique features of the POSIX environment.
miscellaneous functions	provide miscellaneous OpenEdition-related functionality.

The following table lists the SAS/C OpenEdition interfaces. **1003.1a/1003.2** indicates the functions that are defined by the POSIX 1003.1a draft standard or the 1003.2 standard. **Extension** indicates IBM or SAS/C extensions related to POSIX or OpenEdition functionality. **Volume** indicates the volume number of the *SAS/C Library Reference* in which the function is described. Functions that are closely related to ANSI functions or that are useful outside the POSIX/OpenEdition environment are documented in *SAS/C Library Reference, Volume 1.*

Table 19.1 *SAS/C OpenEdition Interfaces*

Function	Extension	10031a/1003.2	Volume
Process Control Functions			
`atfork`	Yes		II
`execl`			II
`execle`			II
`execlp`			II
`execv`			II
`execve`			II
`execvp`			II
`_exit`			II
`fork`			II
`getpid`			II
`getppid`			II
`oeattach`	Yes		II
`oeattache`	Yes		II
`times`			II
`w_getpsent`	Yes		II
`wait`			II
`waitpid`			II
Permission Functions			
`chaudit`	Yes		I
`chmod`			II
`chown`			II
`fchaudit`	Yes		II
`fchmod`		Yes	I
`fchown`		Yes	II
`getegid`			II
`geteuid`			II
`getgid`			II
`getgrgid`			II
`getgrnam`			II
`getgroups`			II

continued

Table 19.1 (*continued*)

Function	Extension	10031a/1003.2	Volume
Permission Functions (*continued*)			
`getgroupsbyname`	Yes		II
`getlogin`			II
`getpwnam`			II
`getpwuid`			II
`getuid`			II
`initgroups`	Yes		II
`__passwd`	Yes		II
`setgid`			II
`setgroups`	Yes		II
`setuid`			II
`umask`			II
Signal Functions			
`alarm`			I
`kill`			I
`ecbsuspend`	Yes		I
`oesigsetup`	Yes		I
`pause`			I
`sigaction`			I
`sigaddset`			I
`sigdelset`			I
`sigemptyset`			I
`sigfillset`			I
`sigismember`			I
`siglongjmp`			I
`sigpending`			I
`sigprocmask`			I
`sigsetjmp`			I
`sigsuspend`			I
`sleep`			I

continued

Table 19.1 (*continued*)

Function	Extension	10031a/1003.2	Volume
I/O Functions			
`close`			I
`creat`			I
`dup`			I
`dup2`			I
`fcntl`			I
`fdopen`			I
`fileno`			I
`fsync`		Yes	I
`ftruncate`		Yes	I
`lseek`			I
`open`			I
`pipe`			I
`read`			I
`w_ioctl`	Yes		I
`write`			I
Directory Functions			
`chdir`			I
`closedir`			I
`getcwd`			I
`mkdir`			I
`opendir`			I
`readdir`			I
`rewinddir`			I
`rmdir`			I
File Utilities			
`access`			I
`fstat`			I
`link`			I
`lstat`		Yes	I
`mkfifo`			I

continued

Table 19.1 (*continued*)

Function	Extension	10031a/1003.2	Volume
File Utilities (*continued*)			
`mknod`	Yes		I
`readlink`		Yes	I
`rename`			I
`stat`			I
`symlink`		Yes	I
`unlink`			I
`utime`			I
Terminal Functions			
`cfgetispeed`			II
`cfgetospeed`			II
`cfsetispeed`			II
`cfsetospeed`			II
`ctermid`			I
`isatty`			I
`tcdrain`			II
`tcflow`			II
`tcflush`			II
`tcgetattr`			II
`tcsendbreak`			II
`tcsetattr`			II
`ttyname`			I
Sessions and Process Groups Functions			
`getpgrp`			II
`setpgid`			II
`setsid`			II
`tcgetpgrp`			II
`tcsetpgrp`			II
Environmental Interfaces			
`clearenv`		Yes	I
`fpathconf`			II

continued

Table 19.1 (*continued*)

Function	Extension	10031a/1003.2	Volume
Environmental Interfaces (*continued*)			
`getdtablesize`	Yes		II
`getenv`			I
`pathconf`			II
`pclose`		Yes	I
`popen`		Yes	I
`setenv`		Yes	I
`sysconf`			II
`system`		Yes	I
`uname`			II
Miscellaneous Functions			
`mount`	Yes		II
`tzset`			I
`umount`	Yes		II
`w_getmntent`	Yes		II
`w_statfs`	Yes		II

SAS/C also provides the following short-cut functions, which invoke OpenEdition functionality, but do not support equivalent SAS/C MVS-oriented functionality:

```
_access
_close
_fcntl
_fsync
_lseek
_open
_read
_rename
_unlink
_write
```

These functions are described in *SAS/C Library Reference, Volume 1.*

MVS Considerations

SAS/C OpenEdition support can be used by several different kinds of programs. A POSIX-conforming program is one that uses only POSIX concepts; it can be written and executed without concern for interactions between the POSIX implementation and the underlying MVS system.

OpenEdition also supports mixed mode programs that combine base MVS and POSIX functionality, for instance, to enable a user shell command to write

a VSAM file or to enable a TSO command to create new processes and communicate with the processes through pipes. The rest of this section discusses the interaction between MVS and POSIX.

fork

When the **fork** system call creates a duplicate of a process, not all of the information is copied to the new address space. Information that is copied includes:

- all non-supervisor subpools
- all dynamically loaded modules
- system error (ESPIE and ESTAE) exits
- most OpenEdition resources, such as open file descriptors and user and group IDs.

Information that is not copied includes:

- all other tasks in the address space.
  The TCB and other system control blocks may also be in different locations in the child and in the parent.
- DD statements.
  The child process executes with no DD statements, except possibly for a STEPLIB.
- Open MVS files.
  The DCBs for open files are copied, but because DD statements and system I/O control blocks are not copied, these DCBs are not valid in the child. Similarly, open non-integrated sockets are not inherited by the child.
- other system exits, such as STIMER exits.

These discrepancies in the information that is duplicated by **fork** has the following consequences:

- No CTRANS DD statement is defined in the child. The library automatically copies any CTRANS DD statement in the parent to the child before it returns control to the child from fork.
- Attempts to access an MVS file opened in the parent from the child will fail. These access attempts are captured by the library without referencing any invalid DCB. Closing a file that is opened in the parent is harmless, unless there is unflushed output data; unflushed output data cannot be written.
- Signal handling for non-OpenEdition asynchronous signals is in an undefined state after **fork**. Most of these signals cannot occur in an address space created by **fork**.

Programs that use **fork** can define **atfork** exits to take the appropriate action before and after a call to **fork**.

exec

When one of the **exec** functions is called (**execl**, **execle**, **execlp**, **execv**, **execve**, or **execvp**), OpenEdition terminates the current process and begins a new process. Every task in the address space is terminated. A new job step is inserted to reinitialize the address space and run the specified program.

The new program inherits many of the OpenEdition attributes of its caller, such as open files, blocked signals, current alarm status, and process id. The new program does not inherit MVS-oriented information such as DD statements (other than STEPLIB) and blocking of non-OpenEdition signals or dynamically loaded modules.

The only allocated storage in the new process is associated with the program's arguments and environment variables. Any information to be passed to a program called by one of the **exec** functions must be passed in argument or environment variables. You may want to consider using the **oeattach** function, rather than **fork** and **exec**, if you want to share storage with a child process.

Because the program called by the **exec** function begins execution with no DD statements, problems can arise while accessing the transient library. If the

SAS/C transient library is not in linklist or LPALIB, you may need to define the ddn_CTRANS environment variable before calling an exec function to execute a SAS/C program. See "Executing C Programs" in *SAS/C Compiler and Library User's Guide* for more information.

Standard Types

The SAS/C implementation of the POSIX.1 standard defines the following standard data types:

Table 19.2 *Standard POSIX Data Types*

Name	Type	Header File	Use parasp=5pt
`cc_t`	`char`	`<termios.h>`	Terminal control character
`clock_t`	`double`	`<time.h>`	Clock tick time unit
`dev_t`	`unsigned int`	`<sys/types.h>`	Device number
`gid_t`	`unsigned int`	`<sys/types.h>`	Group ID
`ino_t`	`unsigned long`	`<sys/types.h>`	File serial number
`mode_t`	`unsigned long`	`<sys/types.h`	File access mode
`nlink_t`	`int`	`<sys/types.h>`	File link count
`off_t`	`long`	`<sys/types.h>`	Type for `lseek` position
`pid_t`	`unsigned int`	`<sys/types.h>`	Process ID or process group ID
`size_t`	`int`	`<sys/types.h>`	Result of `sizeof` operator
`speed_t`	`unsigned char`	`<termios.h>`	Terminal I/O speed
`ssize_t`	`int`	`<sys/types.h>`	Signed equivalent of `size_t`
`tcflag_t`	`int`	`<termios.h>`	Terminal control flag
`time_t`	`double`	`<time.h>`	Elapsed time unit
`uid_t`	`unsigned int`	`<sys/types.h>`	User ID

HFS Files and DDnames

MVS allows you to define DD statements for OpenEdition Hierarchical File System (HFS) files in batch, or by using the TSO ALLOCATE command in TSO. C programs that access files through DDnames can easily access HFS files without recompilation. Access to an HFS file through a DD statement does not change the behavior of the file; a file's characteristics, including buffering, seeking and sharing behavior, are the same whether the file is accessed by name or through a DD statement.

Refer to Chapter 3, "I/O Functions," in *SAS/C Library Reference, Volume 1* for information on using environment variables to replace DD statements in programs called by `exec*`.

Signal Handling and ABENDs

MVS and POSIX have different concepts of abnormal termination. Under MVS, a task terminates abnormally as a result of the ABEND supervisor call. Each ABEND is identified by a numeric code which defines the reason for termination.

According to POSIX, a process terminates abnormally as the result of receiving a signal for which the default action is process termination. Abnormal termination is identified only by the signal name.

From an MVS perspective, any abnormal termination resulting from a POSIX signal causes an ABEND with a system completion code of `EC6`. The signal number can be extracted from the MVS reason code associated with the ABEND.

From a POSIX perspective, the situation is more complex. If a process terminates as the result of an ABEND that is not the direct result of a POSIX signal, the behavior depends on actions of the run-time library. If the run-time library's ABEND handling is not active due to the use of the **=nohtsig** option, or if the library is incapable of handling the ABEND because, for example, library control blocks have been corrupted, the ABEND can complete. OpenEdition interprets the completion by ABEND as termination by **SIGKILL**. If the library can handle the ABEND, the ABEND is transformed into an OpenEdition signal: **SIGABRT** for a user ABEND, or **SIGABND** for a system ABEND. If the program has defined a handler for the signal, control is passed to the handler. Otherwise, the library completes termination by sending the appropriate signal to itself; this causes the process status, as seen by the parent process, to indicate that signal.

When termination results from a program check such as an invalid memory access, the library's traceback may show either the traditional **0Cx** ABEND code or an **EC6** ABEND. For programs that use OpenEdition signal handling, it is difficult to predict which ABEND code will be reported.

Multiple Processes in an Address Space

The SAS/C **oeattach** and **oeattache** functions allow you to create a child process that runs in the same address space as the caller. Processes created in this manner do not obey all the normal POSIX rules for processes. In particular:

- You cannot invoke a **setuid** or **setgid** program unless the program's specified user ID or group ID is the same as the current user ID or group ID.
- When the parent process is terminated, the child process is forced to terminate as well.
- If the child process calls an **exec** function, a new subtask of the parent process is created, that is, the **exec** call does not terminate the address space. An attempt to **exec** a **setuid** or **setgid** program that does not match the current user ID or group ID will fail.
- All processes in an address space share the same DD statements for MVS files. However, MVS files open in the parent process are not inherited by the child process.
- No subpools are shared between a parent process and a child created with **oeattach**.
- If **oeattach** is called from a TSO address space, the child process can read from and write to the user's TSO terminal. However, some other TSO-related functionality is unavailable, such as TSO external scope environment variables and the **tso:** feature of the system function.

Behavior when OpenEdition is not Available

In general, using an OpenEdition interface in a system where OpenEdition is not installed or not running is not harmful. The library issues a diagnostic message, sets **errno** to **EMVSNOTUP**, and returns.

There are a few functions for which the 1003.1 standard does not specify a way to fail because it is impossible for the function to fail in the UNIX environment. For instance, in the UNIX environment, **getpid** cannot fail, because every program has a process ID. In MVS, if OpenEdition is not installed or has failed, it is impossible to obtain a process ID. In these cases, the library issues an ABEND user 1230.

20 POSIX Function Reference

20-1 Introduction

Introduction

This chapter contains a description of each POSIX function in the SAS/C Library, except for those functions that are described in the *SAS/C Library Reference, Third Edition, Volume 2, Release 6.00*.

atfork Define Fork Exits

SYNOPSIS

```
#include <sys/types.h>

int atfork(void *anyData, void(*preExit)(void *),
           void(*parentExit)(void *), void(*childExit)(void *));
```

DESCRIPTION

atfork defines up to three fork exits that are called by the library during the execution of the **fork** function. The arguments to **atfork** are:

anyData
is a **void** pointer that is passed to the three fork exits. This pointer can be used to pass the address of a structure that is then cast to an appropriate type within the exit functions.

preExit
is a pointer to an exit function that is called before the **fork** system call is issued. An address of **0** indicates that no function is defined.

parentExit
is a pointer to an exit function that is called in the parent process after the **fork** is complete. An address of **0** indicates that no function is defined.

childExit
is a pointer to an exit function that is called in the child process after the **fork** is complete. An address of **0** indicates that no function is defined.

Notice that the same pointer, **anyData**, is passed to all three exit functions. This enables the sharing of data between the three functions, and it also enables you to pass information from the function pointed to by **preExit** to either of the functions pointed to by **parentExit** and **childExit**.

atfork exits may be conveniently used to checkpoint and restore data that is not preserved over the fork: for example, names of open MVS files.

RETURN VALUE

atfork returns a **0** if successful and a **-1** if unsuccessful.

EXAMPLE

This example is intended for use in a program that opens a VSAM file named **vsamin**. This program would also use **fork** to create new processes that continue to run the same application. (That is, they do not call **exec**.)

Because VSAM files are not supported in the hierarchical file system, when the **fork** is done the **FILE** pointer referencing the VSAM file will become invalid in the child.

The example uses **atfork** as follows:

1. In the parent, to determine the name of the VSAM file using the **fnm** and **osddinfo** functions.
2. In the child, to reopen the file using the name obtained by step 1.

atfork Define Fork Exits
(continued)

Notice that if the original file name is a DDname, the use of **osddinfo** is necessary, since the child process will not have the DD statement allocated.

Note: This example must not be compiled with the **posix** option.

```
#include <sys/types.h>
#include <unistd.h>
#include <lclib.h>
#include <lcio.h>
#include <string.h>
#include <os.h>

extern FILE *vsamin;

static void pre_parent(void *), post_child(void *);

int setup(void) {
   static char vsamdsn[49];
   return atfork(vsamdsn, &pre_parent, 0, &post_child);
}

static void pre_parent(void *atfork_arg) {
   char *vsamdsn = atfork_arg;
   char *filename;
   int rc;

   if (!vsamin) return;
   filename = fnm(vsamin);         /* Find VSAM file name.         */
   strcpy(vsamdsn, filename);      /* Copy file name to buffer.    */
   strlwr(vsamdsn);
   if (strncmp(vsamdsn, "ddn:", 4) == 0) filename = filename+4;
                                   /* Skip ddn prefix.             */
   else if (strncmp(vsamdsn, "dsn:", 4) == 0 ||
            strncmp(vsamdsn, "tso:", 4) == 0) return;
                                   /* all done if cms or tso file  */
   memcpy(vsamdsn, "dsn:", 4);     /* Make file name dsn form.     */
   rc = osddinfo(filename, vsamdsn+4, NULL, NULL, NULL, NULL);
                                   /* Extract dsname.              */
   if (rc != 0) vsamdsn[0] = '\0'; /* If dsn unavail., remove name.*/
   return;
}
```

atfork Define Fork Exits
(continued)

```
static void post_child(void *atfork_arg) {
   char *vsamdsn = atfork_arg;

   if (*vsamdsn) {                /* If a filename was found.     */
      freopen(vsamdsn, "rk", vsamin);
   }                              /* Reopen vsamin to same file,  */
   else {                         /* else close it                */
      if (vsamin) fclose(vsamin);
      vsamin = 0;                 /* and zero file pointer.       */
   }
   return;
}
```

RELATED FUNCTIONS

fork

cfgetispeed Determine Input Baud Rate

SYNOPSIS

```
#include <termios.h>

speed_t cfgetispeed(const struct termios *termptr);
```

DESCRIPTION

cfgetispeed returns the input baud rate, which is stored in the **termios** structure pointed to by **termptr**. This structure is defined in **<termios.h>** and contains terminal attribute information.

The **tcgetattr** function must be used to obtain a copy of the **termios** structure before you can use the **cfgetispeed** function to extract the input baud rate from the copy of the structure. The **cfsetispeed** and **tcsetattr** functions can then be used to change the input baud rate.

RETURN VALUE

The return value is a defined type, **speed_t**, which is located in **<termios.h>**. Each value is associated with an asynchronous line speed as follows:

Return Value	Baud Rate
B0	hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

cfgetispeed Determine Input Baud Rate
(continued)

PORTABILITY

The **cfgetispeed** function is defined by the POSIX.1 standard and provides portability between operating environments. Note that OpenEdition only supports pseudoterminals and that baud rate does not affect the operation of a pseudoterminal.

EXAMPLE

The following code fragment illustrates the use of **tcgetattr** and **cfgetospeed** to determine the speed of **stdin**.

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   struct termios termAttr;
   speed_t baudRate;
   char *inputSpeed = "unknown";

      /* Obtain a copy of the termios structure for stdin. */
   tcgetattr(STDIN_FILENO, &termAttr);
      /* Get the input speed.                              */
   baudRate = cfgetispeed(&termAttr);
      /* Print input speed.                                */
   switch (baudRate) {
      case B0:      inputSpeed = "none"; break;
      case B50:     inputSpeed = "50 baud"; break;
      case B110:    inputSpeed = "110 baud"; break;
      case B134:    inputSpeed = "134 baud"; break;
      case B150:    inputSpeed = "150 baud"; break;
      case B200:    inputSpeed = "200 baud"; break;
      case B300:    inputSpeed = "300 baud"; break;
      case B600:    inputSpeed = "600 baud"; break;
      case B1200:   inputSpeed = "1200 baud"; break;
      case B1800:   inputSpeed = "1800 baud"; break;
      case B2400:   inputSpeed = "2400 baud"; break;
      case B4800:   inputSpeed = "4800 baud"; break;
      case B9600:   inputSpeed = "9600 baud"; break;
      case B19200:  inputSpeed = "19200 baud"; break;
      case B38400:  inputSpeed = "38400 baud"; break;
   }
   printf("Input speed = %s\n", inputSpeed);
}
```

RELATED FUNCTIONS

cfgetospeed, **cfsetispeed**, **tcgetattr**

cfgetospeed Determine Output Baud Rate

SYNOPSIS

```
#include <termios.h>

speed_t cfgetospeed(const struct termios *termptr);
```

DESCRIPTION

cfgetospeed returns the output baud rate, which is stored in the **termios** structure pointed to by **termptr**. This structure is defined in **<termios.h>** and contains terminal attribute information.

The **tcgetattr** function must be used to obtain a copy of the **termios** structure before you can use the **cfgetospeed** function to extract the output baud rate from the copy of the structure. The **cfsetospeed** and **tcsetattr** functions can then be used to change the output baud rate.

RETURN VALUE

The return value is a defined type, **speed_t**, which is located in **<termios.h>**. Each value is associated with an asynchronous line speed as follows:

Return Value	Baud Rate
B0	hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

cfgetospeed Determine Output Baud Rate
(continued)

PORTABILITY

The **cfgetospeed** function is defined by the POSIX.1 standard and provides portablility between operating environments. Note that OpenEdition only supports pseudoterminals and that baud rate does not affect the operation of a pseudoterminal.

EXAMPLE

The following code fragment illustrates the use of **tcgetattr** and **cfgetospeed** to determine the speed of **stdout**.

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   struct termios termAttr;
   speed_t baudRate;
   char *outputSpeed = "unknown";

      /* Obtain a copy of the termios structure for stdout. */
   tcgetattr(STDOUT_FILENO, &termAttr);
      /* Get the output speed.                              */
   baudRate = cfgetospeed(&termAttr);
      /* Print output speed.                                */
   switch (baudRate) {
      case B0:      outputSpeed = "none"; break;
      case B50:     outputSpeed = "50 baud"; break;
      case B110:    outputSpeed = "110 baud"; break;
      case B134:    outputSpeed = "134 baud"; break;
      case B150:    outputSpeed = "150 baud"; break;
      case B200:    outputSpeed = "200 baud"; break;
      case B300:    outputSpeed = "300 baud"; break;
      case B600:    outputSpeed = "600 baud"; break;
      case B1200:   outputSpeed = "1200 baud"; break;
      case B1800:   outputSpeed = "1800 baud"; break;
      case B2400:   outputSpeed = "2400 baud"; break;
      case B4800:   outputSpeed = "4800 baud"; break;
      case B9600:   outputSpeed = "9600 baud"; break;
      case B19200:  outputSpeed = "19200 baud"; break;
      case B38400:  outputSpeed = "38400 baud"; break;
   }
   printf("Output speed = %s\n", outputSpeed);
}
```

RELATED FUNCTIONS

cfgetispeed, **cfsetospeed**, **tcgetattr**

cfsetispeed Set Input Baud Rate

SYNOPSIS

```
#include <termios.h>

int cfsetispeed(struct termios *terminal, speed_t speed);
```

DESCRIPTION

cfsetispeed is used to set a new input baud rate in the **termios** structure pointed to by **terminal**. This structure is defined in **<termios.h>** and contains terminal attribute information.

The **tcgetattr** function must be used to obtain a copy of the **termios** structure before you can use the **cfsetispeed** function to set the input baud rate. The **cfsetispeed** function changes the baud rate in this copy and the **tcsetattr** functions can then be used to update the **termios** control structure.

The defined type, **speed_t**, specifies the baud rate. Each value is associated with an asynchronous line speed as follows:

Return Value	Baud Rate
B0	hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

RETURN VALUE

If successful, **cfsetispeed** returns **0**. A **-1** is returned if unsuccessful.

cfsetispeed Set Input Baud Rate

(continued)

PORTABILITY

The **cfsetispeed** function is defined by the POSIX.1 standard and provides portability between operating environments. Note that OpenEdition only supports pseudoterminals and that baud rate does not affect the operation of a pseudoterminal.

EXAMPLE

The following example illustrates the use of **cfsetispeed** to set a new output baud rate:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>

main()
{
   struct termios termAttr;
   speed_t baudRate;

      /* Make a copy of the termios structure. */
   tcgetattr(STDIN_FILENO, &termAttr);
      /* Get the input speed.                  */
   baudRate = cfgetispeed(&termAttr);
      /* Set output speed if not 9600 baud.    */
   if (baudRate != B9600) {
      cfsetispeed(&termAttr, B9600);
      tcsetattr(STDIN_FILENO, TCSANOW, &termAttr);
   }
}
```

RELATED FUNCTIONS

cfgetispeed, **cfsetospeed**, **tcsetattr**

cfsetospeed Set Output Baud Rate

SYNOPSIS

```
#include <termios.h>

int cfsetospeed(struct termios *terminal, speed_t speed);
```

DESCRIPTION

cfsetospeed is used to set a new output baud rate in the **termios** structure. (**terminal** points to a copy of this structure.) This structure is defined in **<termios.h>** and contains terminal attribute information.

The **tcgetattr** function must be used to make a copy of the **termios** structure before you can use the **cfsetospeed** function to set the output baud rate. The **cfsetospeed** function changes the baud rate in this copy and the **tcsetattr** functions can then be used to update the **termios** control structure.

The defined type, **speed_t**, specifies the baud rate. Each value is associated with an asynchronous line speed as follows:

Return Value	Baud Rate
B0	hang up
B50	50 baud
B75	75 baud
B110	110 baud
B134	134.5 baud
B150	150 baud
B200	200 baud
B300	300 baud
B600	600 baud
B1200	1200 baud
B1800	1800 baud
B2400	2400 baud
B4800	4800 baud
B9600	9600 baud
B19200	19,200 baud
B38400	38,400 baud

RETURN VALUE

If successful, **cfsetospeed** returns **0**. A **-1** is returned if unsuccessful.

cfsetospeed Set Output Baud Rate

(continued)

PORTABILITY

The **cfsetospeed** function is defined by the POSIX.1 standard and provides portability between operating environments. Note that OpenEdition only supports pseudoterminals and that baud rate does not affect the operation of a pseudoterminal.

EXAMPLE

The following example illustrates the use of **cfsetospeed** to set a new output baud rate:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   struct termios termAttr;
   speed_t baudRate;

      /* Obtain a copy of the termios structure for stdout. */
   tcgetattr(STDOUT_FILENO, &termAttr);
      /* Get the output speed.                              */
   baudRate = cfgetospeed(&termAttr);
      /* Set output speed if not 9600 baud.                 */
   if (baudRate != B9600) {
      cfsetospeed(&termAttr, B9600);
      tcsetattr(STDOUT_FILENO, TCSADRAIN, &termAttr);
   }
}
```

RELATED FUNCTIONS

cfgetospeed, **cfsetispeed**, **tcsetattr**

chaudit Change File Audit Flags (Using Pathname)

SYNOPSIS

```
#include <sys/stat.h>

int chaudit(const char *pathname, int auditFlags, int securityType);
```

DESCRIPTION

chaudit changes the audit flags for the file specified by the **pathname** argument. The audit flags control the type of file requests that the OpenEdition MVS security product audits. The **chaudit** function can change either user audit flags or security auditor audit flags, depending on the setting of the **securityType** argument.

The **auditFlags** argument is formed by ORing any of the following flags, which are defined in **<sys/stat.h>**:

AUDTREDFAIL
audit failing read requests.

AUDTREADSUCC
audit successful read requests.

AUDTWRITEFAIL
audit failing write requests.

AUDWRITESUCC
audit successful write requests.

AUDTEXECFAIL
audit failing search or execute requests.

AUDTEXECSUCC
audit successful search or execute requests.

The flags for the **securityType** argument are also defined in **<sys/stat.h>** and can be either of the following:

AUDT_USER
specifies that the changes should be applied to the user audit flags. In order to change the user audit flags for a file, the user must be either the file owner or have the appropriate authority to make the changes.

AUDT_AUDITOR
specifies that the changes should be applied to the security auditor audit flags. This **securityType** argument can only be specified by a user with security auditor authority.

PORTABILITY

The **chaudit** function is useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

RETURN VALUE

chaudit returns **0** if successful and a **-1** if unsuccessful.

chaudit Change File Audit Flags (Using Pathname)
(continued)

EXAMPLE

The following example illustrates the use of **chaudit** to change a file's user audit flags.

Note: You must specify the **posix** option when compiling this example.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int fd;
   char words[] = "Test File";

      /* Create a test file.              */
   if ((fd = creat("test.file", S_IRUSR|S_IWUSR)) == -1)  {
      perror("creat error");
      exit(1);
   }
   else {
      write(fd, words, strlen(words));
      close(fd);
   }
      /* Change the user audit flags.     */
   if (chaudit("test.file", AUDTREADFAIL|AUDTWRITEFAIL, AUDT_USER) != 0)
      perror("chaudit error");
}
```

RELATED FUNCTIONS

chmod, **chown**, **fchaudit**,

chown Change File Owner or Group (Using Pathname)

SYNOPSIS

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);
```

DESCRIPTION

chown changes the owner or owning group of a file. It can be used with either regular files or special files, such as directories or FIFO files. **pathname** is the name of the file. **owner** is the user ID. **group** is the group ID.

OpenEdition implements a restricted **chown** for all files. A call to **chown** can succeed in only one of two ways:

- the caller is a superuser
- the effective user ID is the same as the owner of the file, which is equal to the owner argument to **chown**, and the group argument is the effective group ID or one of the user's supplementary ID's.

If the **S_IXUSR**, **S_IXGRP**, or **S_IXOTH** bit is set in the file permissions, **chown** clears the **S_ISUID** and **S_ISGID** bits of the permissions.

RETURN VALUE

chown returns **0** if successful and a **-1** if not successful.

EXAMPLE

The following example illustrates the use of **chown** to change a file's owner and group:

```
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
   struct group *billing;
   int rc;

   billing = getgrnam("BILLING");
   if (!billing) {
      perror("getgrnam error");
      abort();
   }
   if (argc <= 1) {
      printf("No file name specified.\n");
      exit(EXIT_FAILURE);
```

chown Change File Owner or Group (Using Pathname)
(continued)

```
        }
        rc = chown(argv[1], geteuid(), billing->gr_gid);
        if (rc == 0) {
           printf("Ownership of %s assigned to BILLING.\n", argv[1]);
           exit(EXIT_SUCCESS);
        }
        else {
           perror("chown error");
           abort();
        }
     }
```

RELATED FUNCTIONS

chmod, **fchown**

execl Overlay Calling Process and Run New Program


SYNOPSIS

```
#include <unistd.h>

int execl(const char *file, const char *arg0, ..., NULL);
```

DESCRIPTION

Like all of the **exec** functions, **execl** replaces the calling process image with a new process image. This has the effect of running a new program with the process ID of the calling process. Note that a new process is not started; the new process image simply overlays the original process image. The **execl** function is most commonly used to overlay a process image that has been created by a call to the **fork** function.

file
is the filename of the file that contains the executable image of the new process.

arg0, ..., NULL
is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a **NULL** pointer. The first argument, **arg0**, is required and must contain the name of the executable file for the new process image. If the new process image is a normal SAS/C **main** program, the list of arguments will be passed to **argv** as a pointer to an array of strings. The number of strings in the array is passed to the **main()** function as **argc**.

ARG_MAX specifies the maximum number of bytes, including the **NULL** terminator at the end of the string, that can be passed as arguments to the new process image. The value of **ARG_MAX** is obtained by calling the **sysconf** function with the **_SC_ARG_MAX** symbol.

RETURN VALUE

A successful call to **execl** does not have a return value because the new process image overlays the calling process image. However, a **-1** is returned if the call to **execl** is unsuccessful.

EXAMPLE

The following code fragment illustrates creating a new process and executing an HFS file called **newShell**:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   pid_t pid;
```

execl Overlay Calling Process and Run New Program
(continued)

```
    if ((pid = fork()) == -1)
       perror("fork error");
    else if (pid == 0) {
       execl("/u/userid/bin/newShell", "newShell", NULL);
       printf("Return not expected. Must be an execl() error.\n");
    }
}
```

RELATED FUNCTIONS

execle, **execlp**, **execv**

execle Overlay Calling Process and Run New Program


SYNOPSIS

```
#include <unistd.h>

int execle(const char *file, const char *arg0, ..., NULL,
           char *const envp[]);
```

DESCRIPTION

Like all of the **exec** functions, **execle** replaces the calling process image with a new process image. This has the effect of running a new progam with the process ID of the calling process. Note that a new process is not started; the new process image simply overlays the original process image. The **execle** function is most commonly used to overlay a process image that has been created by a call to the **fork** function.

file
is the filename of the file that contains the executable image of the new process.

arg0, ..., NULL
is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a **NULL** pointer. The first argument, **arg0**, is required and must contain the name of the executable file for the new process image. If the new process image is a normal SAS/C **main** program, the list of arguments will be passed to **argv** as a pointer to an array of strings. The number of strings in the array is passed to the **main()** function as **argc**.

ARG_MAX specifies the maximum number of bytes, including the **NULL** terminator at the end of the string, that can be passed as arguments to the new process image. The value of **ARG_MAX** is obtained by calling the **sysconf** function with the **_SC_ARG_MAX** symbol.

envp
is a pointer to an array of pointers to null-terminated character strings. A **NULL** pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image. Each string should have the following form:

```
"var=value"
```

RETURN VALUE

A successful call to **execle** does not have a return value because the new process image overlays the calling process image. However, a **-1** is returned if the call to **execle** is unsuccessful.

execle Overlay Calling Process and Run New Program
(continued)

EXAMPLE

The following code fragment illustrates creating a new process and executing an HFS file called **newShell**. The **STEPLIB** environment variable is passed to define a step library for the execution of **newShell**.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   pid_t pid;
   char *const envp[2] = {"STEPLIB=SASC.V6.LINKLIB", NULL};

   if ((pid = fork()) == -1)
      perror("fork error");
   else if (pid == 0) {
      execle("/u/userid/bin/newShell", "newShell", NULL, envp);
      printf("Return not expected. Must be an execle() error.\n");
   }
}
```

RELATED FUNCTIONS

execl, **execlp**

execlp Overlay Calling Process and Run New Program


SYNOPSIS

```
#include <unistd.h>

int execlp(const char *path, const char *arg0, ..., NULL);
```

DESCRIPTION

Like all of the **exec** functions, **execlp** replaces the calling process image with a new process image. This has the effect of running a new program with the process ID of the calling process. Note that a new process is not started; the new process image simply overlays the original process image. The **execlp** function is most commonly used to overlay a process image that has been created by a call to the **fork** function.

path
identifies the location of the new process image within the hierarchical file system (HFS). If the **path** argument contains a slash (**/**), it is assumed that either an absolute or a relative pathname has been specified. If the **path** argument does not contain a slash, the directories specified by the **PATH** environment variable are searched in an attempt to locate the file.

arg0, ..., NULL
is a variable length list of arguments that are passed to the new process image. Each argument is specified as a null-terminated string, and the list must end with a **NULL** pointer. The first argument, **arg0**, is required and must contain the name of the executable file for the new process image. If the new process image is a normal SAS/C **main** program, the list of arguments will be passed to **argv** as a pointer to an array of strings. The number of strings in the array is passed to the **main()** function as **argc**.

ARG_MAX specifies the maximum number of bytes, including the **NULL** terminator at the end of the string, that can be passed as arguments to the new process image. The value of **ARG_MAX** is obtained by calling the **sysconf** function with the **_SC_ARG_MAX** symbol.

RETURN VALUE

A successful call to **execlp** does not have a return value because the new process image overlays the calling process image. However, a **-1** is returned if the call to **execlp** is unsuccessful.

EXAMPLE

The following example illustrates creating a new process and executing an HFS file called **newShell**. The path **/u/userid/bin** is added at the end of the **PATH** environment variable before calling **execlp**.

Note: You must specify the **posix** option when compiling this example.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main()
```

execlp Overlay Calling Process and Run New Program
(continued)

```
{
   pid_t pid;
   char *pathvar;
   char newpath[1000];

   pathvar = getenv("PATH");
   strcpy(newpath, pathvar);
   strcat(newpath, ":u/userid/bin");
   setenv("PATH", newpath);

   if ((pid = fork()) == -1)
      perror("fork error");
   else if (pid == 0) {
      execlp("newShell", "newShell", NULL);
      printf("Return not expected. Must be an execlp error.\n");
   }
}
```

RELATED FUNCTIONS

execl, **execvp**

execv Overlay Calling Process and Run New Program


SYNOPSIS

```
#include <unistd.h>

int execv(const char *file, char *const argv[]);
```

DESCRIPTION

Like all of the **exec** functions, **execv** replaces the calling process image with a new process image. This has the effect of running a new progam with the process ID of the calling process. Note that a new process is not started; the new process image simply overlays the original process image. The **execv** function is most commonly used to overlay a process image that has been created by a call to the **fork** function.

file
is the filename of the file that contains the executable image of the new process.

argv
is a pointer to an array of pointers to null-terminated character strings. A **NULL** pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, **argv[0]**, is required and must contain the name of the executable file for the new process image.

RETURN VALUE

A successful call to **execv** does not have a return value because the new process image overlays the calling process image. However, a **-1** is returned if the call to **execv** is unsuccessful.

EXAMPLE

The following example illustrates the use of **execv** to execute the **ls** shell command:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   pid_t pid;
   char *const parmList[] = {"/bin/ls", "-l", "/u/userid/dirname", NULL};

   if ((pid = fork()) == -1)
      perror("fork error");
   else if (pid == 0) {
      execv("/bin/ls", parmList);
      printf("Return not expected. Must be an execv error.\n");
   }
}
```

execv Overlay Calling Process and Run New Program
(continued)

RELATED FUNCTIONS

execl, **execle**, **execlp**

execve Overlay Calling Process and Run New Program


SYNOPSIS

```
#include <unistd.h>

int execve(const char *file, char *const argv[], char *const envp[]);
```

DESCRIPTION

Like all of the **exec** functions, **execve** replaces the calling process image with a new process image. This has the effect of running a new program with the process ID of the calling process. Note that a new process is not started; the new process image simply overlays the original process image. The **execve** function is most commonly used to overlay a process image that has been created by a call to the **fork** function.

file
is the filename of the file that contains the executable image of the new process.

argv
is a pointer to an array of pointers to null-terminated character strings. A **NULL** pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, **argv[0]**, is required and must contain the name of the executable file for the new process image.

envp
is a pointer to an array of pointers to null-terminated character strings. A **NULL** pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an environment variable to the new process image.

RETURN VALUE

A successful call to **execve** does not have a return value because the new process image overlays the calling process image. However, a **-1** is returned if the call to **execve** is unsuccessful.

EXAMPLE

The following example illustrates the use of **execve** to execute the **ls** shell command. Notice that the **STEPLIB** environment variable is set for the new process.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   pid_t pid;
   char *const parmList[] =
   {"/bin/ls", "-l", "/u/userid/dirname", NULL};
   char *const envParms[2] = {"STEPLIB=SASC.V6.LINKLIB", NULL};
```

execve Overlay Calling Process and Run New Program
(continued)

```
    if ((pid = fork()) ==-1)
       perror("fork error");
    else if (pid == 0) {
       execve("/u/userid/bin/newShell", parmList, envParms);
       printf("Return not expected. Must be an execve error.\n");
    }
}
```

SEE ALSO

execv, **execvp**

execvp Overlay Calling Process and Run New Program


SYNOPSIS

```
#include <unistd.h>

int execvp(const char *path, char *const argv[]);
```

DESCRIPTION

Like all of the **exec** functions, **execvp** replaces the calling process image with a new process image. This has the effect of running a new program with the process ID of the calling process. Note that a new process is not started; the new process image simply overlays the original process image. The **execvp** function is most commonly used to overlay a process image that has been created by a call to the **fork** function.

path
identifies the location of the new process image within the hierarchical file system (HFS). If the **path** argument contains a slash (**/**), it is assumed that either an absolute or a relative pathname has been specified. If the **path** argument does not contain a slash, the directories specified by the **PATH** environment variable are searched in an attempt to locate the file.

argv
is a pointer to an array of pointers to null-terminated character strings. A **NULL** pointer is used to mark the end of the array. Each character string pointed to by the array is used to pass an argument to the new process image. The first argument, **argv[0]**, is required and must contain the name of the executable file for the new process image.

RETURN VALUE

A successful call to **execvp** does not have a return value because the new process image overlays the calling process image. However, a **-1** is returned if the call to **execvp** is unsuccessful.

EXAMPLE

The following example illustrates the use of **execvp** to execute the **ls** shell command:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   pid_t pid;
   char *const parmList[] =
   {"/bin/ls", "-l", "/u/userid/dirname", NULL};
```

execvp Overlay Calling Process and Run New Program
(continued)

```
    if ((pid = fork()) == -1)
       perror("fork() error");
    else if (pid == 0) {
       execvp("ls", parmList);
       printf("Return not expected. Must be an execvp() error.\n");
    }
}
```

RELATED FUNCTIONS

execlp, **execv**

_exit End Process and Skip Cleanup


SYNOPSIS

```
#include <unistd.h>

void _exit(int status);
```

DESCRIPTION

_exit ends the current process. **status** is the return status for the process.

_exit closes open file descriptors and directory streams in the calling process. If the parent of the calling process is suspended by **wait** or **waitpid**, the low-order eight bits of **status** are available to the parent. Otherwise, the parent process saves the value of **status** to return to the parent in the event of **wait** or **waitpid**. A **SIGCHLD** signal is sent to the parent process.

_exit does not directly terminate child processes. Child processes that continue after the parent process ends receive a new parent process ID of 1.

_exit does not perform C run-time library cleanup; standard I/O stream buffers are not flushed, and **atexit** routines are not called.

RETURN VALUE

_exit is always successful. It does not return a value.

EXAMPLE

The following code fragment illustrates the use of **_exit**:

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

.
.
.

fflush(NULL);
_exit(0);

.
.
.
```

RELATED FUNCTIONS

exit

fchaudit Change File Audit Flags (Using File Descriptor)

SYNOPSIS

```
#include <sys/stat.h>

int fchaudit(int fileDescriptor, int auditFlags, int securityType);
```

DESCRIPTION

fchaudit changes the audit flags for the file specified by the **fileDescriptor** argument. The audit flags control the type of file requests that the OpenEdition MVS security product audits. The **fchaudit** function can change either user audit flags or security auditor audit flags, depending on the setting of the **securityType** argument.

The **auditFlags** argument is formed by ORing any of the following flags, which are defined in **<sys/stat.h>**:

AUDTREDFAIL
audit failing read requests.

AUDTREADSUCC
audit successful read requests.

AUDTWRITEFAIL
audit failing write requests.

AUDWRITESUCC
audit successful write requests.

AUDTEXECFAIL
audit failing search or execute requests.

AUDTEXECSUCC
audit successful search or execute requests.

The flags for the **securityType** argument are also defined in **<sys/stat.h>** and can be either of the following:

AUDT_USER
specifies that the changes should be applied to the user audit flags. In order to change the user audit flags for a file, the user must be either the file owner or have the appropriate authority to make the changes.

AUDT_AUDITOR
specifies that the changes should be applied to the security auditor audit flags. This **securityType** argument can only be specified by a user with security auditor authority.

PORTABILITY

The **fchaudit** function is useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

RETURN VALUE

fchaudit returns **0** if successful and **-1** if unsuccessful.

fchaudit Change File Audit Flags (Using File Descriptor)
(continued)

EXAMPLE

The following example illustrates the use of **fchaudit** to change a file's user audit flags.

Note: You must specify the **posix** option when compiling this example.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>

main()
{
   int fd;
   char words[] = "Test File";

      /* Create a test file.          */
   if ((fd = creat("test.file", S_IRUSR|S_IWUSR)) == -1) {
      perror("creat error");
      _exit(1);
   }
   else
      write(fd, words, strlen(words));

      /* change the user audit flags. */
   if (fchaudit(fd, AUDTREADFAIL|AUDTWRITEFAIL, AUDT_USER) != 0)
      perror("fchaudit error");
   close(fd);
}
```

RELATED FUNCTIONS

chaudit, **fchmod**, **fchown**

fchown Change File Owner or Group (Using File Descriptor)


SYNOPSIS

```
#include <unistd.h>

int fchown(int fileDescriptor, uid_t owner, gid_t group);
```

DESCRIPTION

fchown changes the owner or owning group of a file. It can be used with either regular files or special files, such as directories or FIFO files. **fileDescriptor** is the file descriptor for the file. **owner** is the user ID. **group** is the group ID.

OpenEdition implements a restricted **fchown** for all files. A call to **fchown** can succeed in only one of two ways:

- the caller is a superuser
- the effective user ID is the same as the owner of the file, which is equal to the owner argument to **fchown**, and the group argument is the effective group ID or one of the user's supplementary ID's.

Because **_POSIX_CHOWN_RESTRICTED** is defined for OpenEdition in **<unistd.h>** with a value of **1**, a process can change the group only if the process has the appropriate privileges or its effective user ID is the same as the user ID of the file **owner** and **group** is the effective group ID or one of its supplementary group IDs.

If **fileDescriptor** refers to a regular file and an **S_IXUSR**, **S_IXGRP**, or **S_IXOTH** bit is set in file permissions, **fchown** clears the **S_ISUID** and **S_ISGID** bits of the permissions and returns successfully. If **fileDescriptor** refers to a special file and an **S_IXUSR**, **S_IXGRP**, or **S_IXOTH** bit is set, **fchown** clears the **S_ISUID** and **S_ISGID** bits of the file.

PORTABILITY

The **fchown** function is defined by the POSIX.1a draft standard.

RETURN VALUE

fchown returns **0** if successful and a **-1** if not successful.

EXAMPLE

The following example illustrates the use of **fchown** to change a file's owning group ID.

Note: You must specify the **posix** option when compiling this example.

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <stdio.h>
#include <sys/stat.h>
#include <grp.h>
```

fchown Change File Owner or Group (Using File Descriptor)
(continued)

```
main()
{
   int fd;
   gid_t grpid;
   struct stat fileStatus;
   struct group *gr_data;
   char words[] = "Test File";

      /* Create a test file.                                    */
   if ((fd = creat("test.file", S_IRUSR|S_IWUSR)) == -1) {
      perror("creat error");
      _exit(1);
   }
   else
      write(fd, words, strlen(words));

      /* Get group 10 for new group.                            */
   gr_data = getgrnam("QA");
   if (gr_data == NULL) {
      perror("getgrnam error");
      _exit(1);
   }
   grpid = gr_data->gr_gid;

      /* Get file status and then print user and group IDs. */
   if (fstat(fd ,&fileStatus) != 0)  {
      perror("fstat error");
      _exit(1);
   }
   else  {
      printf("UID=%d GID=%d \n", (int) fileStatus.st_uid,
              (int) fileStatus.st_gid);
        /* Change file ownership.                               */
      if (fchown(fd, seteuid(), grpid) != 0)
         perror("fchown error");
      printf("UID=%d GID=%d \n", (int) fileStatus.st_uid,
              (int) fileStatus.st_gid);
   }
close(fd);
}
```

SEE ALSO

chown, **fchmod**

fork Create a New Process


SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

DESCRIPTION

fork creates a child process that duplicates the process that calls **fork**. The child process is mostly identical to the calling process; however, it has a unique process ID. Any open file descriptors are shared with the parent process. The child process executes independently of the parent process.

Note: For a more complete description of the realationships between the parent process and the child process, see the POSIX 1003.1 standard.

RETURN VALUE

fork returns `0` to the child process and the process ID of the child process to the parent process. **fork** returns `-1` to the parent if it is not successful and does not create a child process.

CAUTION

The MVS attributes of the parent process are generally not inherited by the child. In particular, MVS files open in the parent cannot be accessed in the child. If the parent process were running under TSO, the child process is not able to read or write the TSO terminal. Also, the child process will have no allocated DD statements, except possibly for a STEPLIB.

EXAMPLE

The following example shows a program that uses **fork** to create a second process that communicates with the parent process with a pipe. Both processes send messages to the terminal. The **kill** function is used by the child process to terminate both the child and the parent.

```
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <stdio.h>

main()
{
   int pipefd[2];
   char buf[60];
   int l;
   sigset_t shield;
   pid_t mypid;
```

fork Create a New Process
(continued)

```
   pipe(pipefd);
   if (fork()) {
      sigemptyset(&shield);
      sigaddset(&shield, SIGTERM);
      sigprocmask(SIG_BLOCK, &shield, 0);
      mypid = getpid();
      for(;;) {
         printf("%x: What should I give my true love?\n", mypid);
         l = read(0, buf, 59);
         buf[l-1] = 0;
         write(pipefd[1], buf, l);
         printf("%x: I gave my true love a%s %s.\n", mypid,
                 strchr("aeiou", buf[0])? "n": "", buf);
         if (strstr(buf, "poison") == buf) {
            printf("%x: Oh! The shame!\n", mypid);
            sleep(1);
            break;
         }
         sleep(1);
      }
      sigprocmask(SIG_UNBLOCK, &shield, 0);
      sleep(10);
      kill(mypid, SIGABRT);
   }
   else {
      mypid = getpid();
      for(;;) {
         char ch;
         int i = 0;
         do {
            l = read(pipefd[0], &ch, 1);
            buf[i++] = ch;
         } while(l == 1 && ch);
         printf("%x: My true love gave me a%s %s!\n",
                 mypid, strchr("aeiou", buf[0])? "n": "",
                 buf);
         if (strstr(buf, "poison") != buf) {
            printf("%x: How sweet!\n", mypid);
         } else {
            printf("%x: Oh, yuck!\n", mypid);
            kill(getppid(), SIGTERM);
            printf("%x: Die, oh unfaithful one!\n", mypid);
            kill(mypid, SIGABRT);
         }
      }
   }
   return(255);
}
```

fork Create a New Process
(continued)

Output

Here is a possible terminal transcript from this example program:

```
131073: What should I give my true love?
cherry
131073: I gave my true love a cherry.
131073: What should I give my true love!
262148: My true love gave me a cherry!
262148: How sweet!
poison pomegranate
131073: I gave my true love a poison pomegranate.
131073: Oh! The shame!
262148: My true love gave me a poison pomegranate!
262148: Oh, yuck!
262148: Die, oh unfaithful one!
131073   [15]  Terminated
```

RELATED FUNCTIONS

atfork, **ATTACH**, **exec**, **popen**, **system**

fpathconf Determine Pathname Variables

SYNOPSIS

```
#include <unistd.h>

long fpathconf(int filedes, int configvar);
```

DESCRIPTION

fpathconf determines the value of a configuration variable that is associated with a file descriptor. **filedes** is the file descriptor. **configvar** is the configuration variable.

configvar can be one of the following symbols that are defined in **<unistd.h>**. The values that **fpathconf** returns are listed with each symbol.

_PC_LINK_MAX returns the maximum number of links possible for the file. **filedes** references a directory, and **fpathconf** returns the maximum number of links possible for the directory.

_PC_MAX_CANON returns the maximum number of bytes in a terminal canonical input line. **filedes** refers to a character special file for a terminal.

_PC_MAX_INPUT returns the minimum number of bytes for which space is available in a terminal input queue. This number is also the maximum number of bytes that a user can enter before a portable application reads the input. **filedes** refers to a character special file for a terminal.

_PC_NAME_MAX returns the maximum number of characters in a filename. This number does not include a terminating **NULL** for filenames stored as strings. This limit is for names that do not specify any directories.

_PC_PATH_MAX returns the maximum number of characters in a complete pathname. This number does not include a terminating **NULL** for pathnames stored as strings.

_PC_PIPE_BUF returns the maximum number of bytes that can be written atomically to a pipe. If this number is exceeded, the operation can use more than one physical read and write operation to write and read the data. For FIFO special files, **fpathconf** returns the value for the file itself. For directories, **fpathconf** returns the value for FIFO special files that exist or can be created under the specified directory. For other kinds of files, an **errno** of **EINVAL** is stored.

_PC_CHOWN_RESTRICTED returns whether the use of **chown** is restricted. For directories, **fpathconf** returns the value for files, but not necessarily for subdirectories.

fpathconf Determine Pathname Variables
(continued)

_PC_NO_TRUNC	returns whether or not a filename that is larger than the maximum name size, is considered invalid, or is truncated. For directories, this applies to all files in the directory.
_PC_VDISABLE	returns the character, if any, which can be used to disable special character functions for a terminal file. **filedes** must refer to a character special file for the terminal.

RETURN VALUE

If successful, **fpathconf** returns the value defined above for the particular value of **configvar**. If not successful, **fpathconf** returns **-1**.

EXAMPLE

The following code fragment illustrates the use of **fpathconf**:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

.
.
.
long result;
char fn[]="temp.file";
int fd;
.
.
.
puts("What is the NAME_MAX limit for the current directory?");
if ((result = fpathconf(fd,_PC_NAME_MAX)) == -1)
   if(errno == 0)
      puts("There is no limit.");
   else perror("fpathconf() error");
else
   printf("NAME_MAX is %d\n", result);
.
.
.
```

RELATED FUNCTIONS

pathconf, **sysconf**

getegid Determine Effective Group ID


SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

gid_t getegid(void);
```

DESCRIPTION

getegid determines the effective group ID of the calling process.

RETURN VALUE

If successful, the group ID of the calling process is returned. Under unusual conditions (for instance, if OpenEdition is not running) **getegid** can fail. In this case, **getegid** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **getegid**:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

.
.
.

printf("%d is my group ID.\n", (int) getegid());

.
.
.
```

RELATED FUNCTIONS

geteuid, **getgid**, **setegid**, **setgid**

geteuid Determine Effective User ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

uid_t geteuid(void);
```

DESCRIPTION

geteuid determines the effective user ID of the calling process.

RETURN VALUE

If successful, **geteuid** returns the effective user ID of the calling process. Under unusual conditions (for instance, if OpenEdition is not running) **getegid** can fail. In this case, **geteuid** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **geteuid** to determine the effective user ID of a calling process:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

main()
{
   uid_t uid, euid;

   uid = getuid();
   euid = geteuid();
   printf("The effective user id is  %d\n", (int) geteuid());
   printf("The real user id is  %d\n", (int) getuid());
   if (uid == euid)
      printf("Effective user id and Real user id are the same\n");
}
```

RELATED FUNCTIONS

getegid, **getuid**, **seteuid**, **setuid**

getgid Determine Real Group ID


SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

gid_t getgid(void);
```

DESCRIPTION

getgid determines the real group ID of the calling process.

RETURN VALUE

If successful, **getgid** returns the real group ID of the calling process. Under unusual conditions (for instance, if OpenEdition is not running) **getegid** can fail. In this case, **getgid** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **getgid** to determine the real group ID of a calling process:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

.
.
.

printf("The group ID is %d\n", (int) getgid());

.
.
.
```

RELATED FUNCTIONS

getegid, **getuid**, **setgid**

getgrgid Access Group Database by ID

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrgid(gid_t gid);
```

DESCRIPTION

getgrgid provides information about a group and its members. **gid** is the group ID. **getgrgid** returns a pointer to a **group** structure defined in **<grp.h>** that contains an entry from the group database. **group** contains the following members:

gr_name	name of the group
gr_gid	numerical group ID
gr_mem	null-terminated vector of pointers to the member names.

RETURN VALUE

getgrgid returns a pointer to a **group** structure if successful. **getgrgid** returns a **NULL** pointer if unsuccessful.

Note: The pointer returned by **getgrgid** may be a static data area that can be rewritten by the next call to **getgrgid** or **getgrnam**.

EXAMPLE

The following code fragment illustrates the use of **getgrgid** to obtain a group name:

```
#include <sys/types.h>
#include <stdio.h>
#include <grp.h>
#include <sys/stat.h>

.
.
.

struct stat info;
struct group *grp;

.
.
.
```

getgrgid Access Group Database by ID

(continued)

```
if ((grp = getgrgid(info.st_gid)) == NULL)
   perror("getgrgid() error");
else
   printf("The group name is %sn", grp->gr_name);

.
.
.
```

RELATED FUNCTIONS

getgrnam, **getgroups**, **getpwuid**

getgrnam Access Group Database by Name


SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

struct group *getgrnam(const char *name);
```

DESCRIPTION

getgrnam returns a pointer to a **group** structure defined in **<grp.h>** that contains an entry from the group database. **name** is the entry. **group** contains the following members:

gr_name	name of the group
gr_gid	numerical group ID
gr_mem	null-terminated vector of pointers to the member names.

RETURN VALUE

getgrnam returns a pointer to a **group** structure if successful. Note that the pointer returned by **getgrnam** may be a static data area that can be rewritten by the next call to **getgrgid** or **getgrnam**. **getgrnam** returns a **NULL** pointer if unsuccessful.

EXAMPLE

The following code fragment illustrates the use of **getgrnam** to obtain a list of group members:

```
#include <sys/types.h>
#include <stdio.h>
#include <grp.h>
#include <sys/stat.h>
.
.
.
struct group *grp;
char grpname[]="BILLING";
char  **gmem;
.
.
.
if ((grp = getgrnam(grpname)) == NULL)
   perror("getgrnam() error");
else {
   printf("These are the members of group %s:\n", grpname);
   for (gmem=grp->gr_mem; *gmem != NULL; gmem++)
      printf(" %s\n", *gmem);
}
.
.
.
```

getgrnam Access Group Database by Name
(continued)

RELATED FUNCTIONS

`getgrgid`, **`getpwnam`**

getgroups Determine Supplementary Group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int getgroups(int listSize, gid_t *groupList);
```

DESCRIPTION

getgroups stores the supplementary group IDs of the calling process in an array.

listSize
is the number of elements that can be stored in the array.

groupList
is a pointer to the beginning of the array used to store the supplementary user IDs.

Note: Note that **groupList** is declared as **gid_t *groupList** so that NULL may be passed to **groupList** when **listSize** is **0**. This could not be done if **groupList** was declared as an array.

RETURN VALUE

getgroups returns the number of supplementary group IDs in **groupList**. This value is always less than or equal to the value of **NGROUPS_MAX** defined in **<limits.h>**. If **listSize** is **0**, **getgroups** returns the total number of supplementary group IDs and does not store the group IDs in an array. **getgroups** returns a **-1** if unsuccessful.

EXAMPLE

The following example illustrates the use of **getgroups** to determine supplementary group IDs.

Note: You must specify the **posix** option when compiling this example.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <limits.h>


main()
{
   gid_t groupIDs[NGROUPS_MAX];
   int i, count;

   if ((count = getgroups(NGROUPS_MAX, groupIDs)) == -1)
      perror("getgroups error");
   else
      for (i = 0; i < count; i++)
         printf("Group ID %d = %d\n", i + 1, (int) groupIDs[i]);
}
```

getgroups Determine Supplementary Group IDs
(continued)

RELATED FUNCTIONS

`getgroupsbyname`

getgroupsbyname Determine Supplementary Group IDs for a User Name

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int getgroupsbyname(char * userID, int listSize, gid_t groupList[]);
```

DESCRIPTION

getgroupsbyname stores the supplementary group IDs for a specified user into an array.

userID
identifies the user.

listSize
is the number of elements that can be stored in the array.

groupList
is a pointer to the array used to store the supplementary user IDs.

RETURN VALUE

getgroupsbyname returns **1** plus the number of supplementary group IDs in **groupList**. This value is always less than or equal to the value of **NGROUPS_MAX** defined in **<limits.h>**. If **listSize** is **0**, **getgroups** returns the total number of supplementary group IDs and does not store the group IDs in an array. **getgroups** returns a **-1** if unsuccessful.

PORTABILITY

The **getgroupsbyname** function is not defined by the POSIX.1 standard and should not be used in portable applications.

EXAMPLE

The following example illustrates the use of **getgroupsbyname** to determine the groups that a user belongs to.

Note: You must specify the **posix** option when compiling this example.

```
#include <sys/types.h>
#include <unistd.h>
#include <grp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main(int argc, char *argv[])
{
   gid_t *groupIDs;
   int maxGroups;
   int argGroupID;
   struct group *argGroup;
   int i;
   int found;
```

getgroupsbyname Determine Supplementary Group IDs for a User Name *(continued)*

```
   if (argc < 3) {
      fprintf(stderr, "Two arguments required: username groupname\n");
      abort();
   }

   maxGroups = getgroupsbyname(argv[1], 0, NULL);
                      /* determine number of supplemental groups */
   if (maxGroups <= 0) {
      perror("getgroupsbyname error");
      abort();
   }

   groupIDs = malloc(maxGroups * sizeof(gid_t));
   if (!groupIDs) abort();
   maxGroups = getgroupsbyname(argv[1], maxGroups, groupIDs);
   if (maxGroups <= 0) {
      perror("getgroupsbyname error");
      abort();
   }

   argGroup = getgrnam(argv[2]);    /* look up group name */
   if (!argGroup) {
      perror("getgrnam error");
      abort();
   }
   argGroupID = argGroup->gr_gid;

   found = 0;
   for (i = 0; i < maxGroups; ++i) {
      if (groupIDs[i] == argGroupID) {
         found = 1;
         break;
      }
   }
   if (found) printf("Group %s was found for user %s\n",
                     argv[2], argv[1]);
   else printf("Group %s was not found for user %s\n",
               argv[2], argv[1]);
}
```

RELATED FUNCTIONS

getgroups

getpgrp Determine Process Group ID


SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
```

DESCRIPTION

getpgrp determines the process group ID of the calling process.

RETURN VALUE

If successful, **getpgrp** returns the process group ID of the calling process. Under unusual conditions (for instance, if OpenEdition is not running) **getpgrp** can fail. In this case, **getpgrp** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **getgprp** to determine the process group ID of the calling process:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

.
.
.
if (fork() == 0) {
  printf("The child's PID is %d.  The process group ID is %d\n",
          (int) getpid(), (int) getpgrp());
  exit(0);
}
.
.
.
```

Note: Also see the **setpgid** example.

RELATED FUNCTIONS

setpgid, **setsid**, **tcgetpgrp**

getpid Determine Process ID


SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

DESCRIPTION

getpid determines the process ID of the calling process.

RETURN VALUE

If successful, **getpid** returns the process ID of the calling process. Under unusual conditions (for instance, if OpenEdition is not running) **getpid** can fail. In this case, **getpid** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **getpid** to determine the process ID of the calling process:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

.
.
.
if (fork() == 0) {
   printf("The child's PID is %d.  The process group ID is %dn",
          (int) getpid(), (int) getpgrp());
   exit(0);
}
.
.
.
```

Note: Also see the **setpgid** example.

RELATED FUNCTIONS

getppid

getppid Determine Parent Process ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid(void);
```

DESCRIPTION

getppid determines the process ID of the parent process.

RETURN VALUE

If successful, **getppid** returns the process ID of the parent process. Under unusual conditions (for instance, if OpenEdition is not running) **getppid** can fail. In this case, **getppid** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **getppid** to determine the process ID of a parent process:

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sdtlib.h>

.
.
.
if (fork() == 0) {
   printf("Child process is sending SIGUSR2 to pid %d\n",
           (int) getppid());
   kill(getppid(), SIGUSR2);
   exit(0);
}
.
.
.
```

Note: Also see the **setpgid** example.

RELATED FUNCTIONS

getpid

getpwnam Access User Database by User Name

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwnam(const char *name);
```

DESCRIPTION

getpwnam returns a pointer to the **passwd** structure. **passwd** contains an entry from the user database. **name** is a pointer to the user name.

passwd is defined in **<pwd.h>**. It contains the following members:

pw_name	user name
pw_uid	user ID number
pw_dir	initial working directory
pw_shell	initial user program.

RETURN VALUE

getpwnam returns a pointer to a user database entry if successful. Note that the pointer returned by **getpwnam** may be a static data area that can be rewritten by the next call to **getpwnam** or **getpwuid**. **getpwnam** returns a **NULL** pointer if unsuccessful.

EXAMPLE

The following code fragment illustrates the use of **getpwnam** to obtain the home directory of the user:

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>

.
.
.
struct passwd *p;
char user[]="YVONNE";

if ((p = getpwnam(user)) == NULL)
   perror("getpwnam() error");
else {
   printf("pw_name : %s\n", p->pw_name);
   printf("pw_dir  : %d\n", p->pw_dir);
}
.
.
.
```

getpwnam Access User Database by User Name
(continued)

SEE ALSO

getgrnam, **getpwuid**

getpwuid Access User Database by User ID

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

struct passwd *getpwuid(uid_t uid);
```

DESCRIPTION

getpwuid returns a pointer to the **passwd** structure. **passwd** maps an entry in the user database. **uid** is the user ID for which information is to be returned.

passwd is defined in **<pwd.h>**. It contains the following members:

pw_name	user name
pw_uid	user ID number
pw_dir	initial working directory
pw_shell	initial user program.

RETURN VALUE

getpwuid returns a pointer to the user database entry if successful. Note that the pointer returned by **getpwuid** may be a static data area that can be rewritten by the next call to **getpwnam** or **getpwuid**. **getpwuid** returns a **NULL** pointer if unsuccessful.

EXAMPLE

The following code fragment illustrates the use of **getpwuid** to obtain a pointer to a **passwd** structure:

```
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
.
.
.

struct passwd *p;
uid_t uid=0;

if ((p = getpwuid(uid)) == NULL)
   perror("getpwuid() error");
else {
   printf("getpwuid returned the following name and directory for
      user ID %d:n", (int) uid);
   printf("pw_name : %sn", p->pw_name);
   printf("pw_dir  : %dn", p->pw_dir);
}
.
.
.
```

getpwuid Access User Database by User ID
(continued)

RELATED FUNCTIONS

getgrgid, **getpwnam**

getuid Determine Real User ID


SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void);
```

DESCRIPTION

getuid determines the real user ID of the calling process.

RETURN VALUE

If successful, **getuid** returns the real user ID of the calling process. Under unusual conditions (for instance, if OpenEdition is not running) **getuid** can fail. In this case, **getuid** issues an MVS user ABEND 1230 to indicate the error.

EXAMPLE

The following code fragment illustrates the use of **getuid** to determine the real user ID of a calling process. The user ID is then passed to the **getpwuid** function to obtain a pointer to the user's **passwd** structure.

```
#include <sys/types.h>
#include <pwd.h>
#include <unistd.h>
#include <stdio.h>
.
.
.

struct passwd *p;
uid_t uid;

if ((p = getpwuid(uid = getuid())) == NULL)
   perror("getpwuid() error");
else {
   printf("getpwuid returned the following name and directory for
      your user IDn", (int) uid);
   printf("pw_name : %s\n", p->pw_name);
   printf("pw_dir  : %d\n", p->pw_dir);
}
.
.
.
```

RELATED FUNCTIONS

geteuid, **getgid**, **setuid**

initgroups Initialize Supplementary Group IDs for a Process

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

int initgroups(const char *userID, gid_t groupID);
```

DESCRIPTION

initgroups initializes the supplementary group ID list for a process to a user's supplementary group list. The user is identified by the **userID** argument, which is a 1 to 8 character, null-terminated, MVS user ID. The group ID specified by the **groupID** argument is added to the supplementary list for the process.

Note: This function can only be called by a superuser.

RETURN VALUE

initgroups returns a **0** if successful and a **-1** if unsuccessful.

RELATED FUNCTIONS

getgroupsbyname, **setgroups**

mknod Create a Character or FIFO Special File

SYNOPSIS

```
#include <sys/stat.h>

int mknod(const char *path, mode_t mode, rdev_t device_id);
```

DESCRIPTION

mknod creates a new character special file or a FIFO special file. The **path** argument specifies the pathname of the special file, and the **mode** argument determines which type of special file is created. The following symbols can be specified as the **mode** argument:

S_IFCHR	Character special file
S_IFFIFO	FIFO special file

A character special file is usually associated with a device, and a FIFO (first-in-first-out) special file is a named pipe that is used for communciation between two processes.

The file permissions bits of the new special file can also be initialized with the **mode** argument. To do this, use a bitwise OR operation to combine any of the permissions symbols described for the **mode** argument of the **chmod** function with either **S_IFCHR** or **S_IFFIFO**.

The **device_id** argument identifies the specific device associated with a character special file. This argument is not used with FIFO special files.

The **device_id** argument is one word long, with the high-order 16 bits used to identify the device driver for a class of devices, such as interactive terminals. See IBM's *Application Callable Services for OpenEdition MVS* (SC23-3020) for an explanation of device IDs.

RETURN VALUE

mknod returns **0** if successful and **-1** if it is unsuccessful.

PORTABILITY

The **mknode** function is useful in POSIX applications; however, it is not defined by POSIX.1 and should not be used in strictly conforming applications. POSIX.1 does not provide a way to make character special files.

EXAMPLE

The following code fragment illustrates the use of **mknod** to create a character special file with user read and write permissions set and a **device_id** of **0x00020003**. (It is a slave pseudo TTY.)

```
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
```

mknod Create a Character or FIFO Special File
(continued)

```
#define deviceClass 0x00020000
#define ttyNumber 3
.
.
.
char charSpecial[]="pseudo.tty";
.
.
.
mknod(charSpecial, S_IFCHR|S_RUSR|S_IWUSR, deviceClass|ttyNumber);
.
.
.
```

RELATED FUNCTIONS

mkfifo

mount Mount a File System

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mount(const char *mountPoint, char *fileSystem, char *fileSysType,
          mtm_t mountMode, int parmLength, char *parm);
```

DESCRIPTION

mount specifies a mount point for a file system, making it available within the hierarchical file system. Mounts must be requested by a superuser, and only one mount point may be specified for each file system.

mountPoint
specifies the mount point directory.

fileSystem
specifies the null-terminated name of the file system to be mounted. **fileSystem** is a 1- to 44-character MVS data set name that is specified as all uppercase letters.

fileSysType
specifies the type of the file system. The type is a maximum length of 8 characters and must match the TYPE operand of a FILESYSTYP statement in the BPXPRMxx parmlib member for the file system. The normal **fileSysType** is **''HFS''**. Refer to *MVS/ESA Initialization and Tuning Reference* (SC28-1452) for more information on BPXPRMxx.

mountMode
determines the file system mode:

MTM_RDONLY	Read-only file system
MTM_RDWR	Read/Write file system

parmLength
specifies the length of the **parm** argument, up to a maximum of 1024 characters.

parm
is a parameter that is passed to the mounting file system specified by **fileSysType**. The content and format of **parm** is determined by the file system.

The **parmLength** and **parm** parameters are ignored when mounting a hierarchical file system (HFS) data set.

RETURN VALUE

mount returns **0** if the mount is successful and a **-1** if it fails.

mount Mount a File System
(continued)

PORTABILITY

The **mount** function may be useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

EXAMPLE

The following code fragment illustrates the use of **mount** to establish a mount point in the hierarchical file system.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

main()
{
   char *mountPoint = "/usr";
   char *HFS = "POSIX.FILE.SYSTEM";
   char *mountType[9] = "HFS     ";
   .
   .
   .
   if (mount(mountPoint, HFS, mountType, MTM_RDWR, 0, NULL) != 0)
      perror("error mounting file system");
   .
   .
   .
}
```

RELATED FUNCTIONS

umount, **w_getmntent**, **w_statfs**

oeattach Create a Child Process as a Subtask

SYNOPSIS

```
#include <sys/types.h>
#include <lclib.h>

pid_t oeattach(const char *file, char *const argv[]);
```

DESCRIPTION

The **oeattach** function creates a new process in the address space of the caller. In MVS terms, the new process runs as a subtask of the calling process. **oeattach** can be used as a high-performance substitute for **fork** followed by **execv**. It also provides a way for two processes to communicate using shared memory.

The **file** argument identifies the HFS file to be executed in the new process. The **argv** argument is a pointer to an array of pointers to null-terminated strings, to be passed as arguments to the new process. A **NULL** pointer marks the end of the array. The first argument, **argv[0]**, is required, and must contain the name of the executable file for the new process.

The environment variables for the new process are the same as the program scope environment variables for the calling process. Also, the new process inherits other attributes (for example, signal mask) from its caller as described for the **execl** function.

RETURN VALUE

If **oeattach** is successful, it returns the process ID of the new process. If it is unsuccessful, it returns **-1**.

CAUTIONS

Because the process created by **oeattach** is an MVS subtask of the caller, this process is abnormally terminated if the calling process terminates.

Certain restrictions apply in the use of **oeattach**. For instance, it may not be used to invoke a setuid program. See the description of the BPX1ATX service routine in the Assembler Callable Services for OpenEdition MVS publication (SC23-3020) for further information.

Processes created in TSO by use of **oeattach** have only partial access to TSO facilities. They can read and write to the TSO terminal and handle TSO attention signals. They cannot use the **system** function to invoke TSO commands, use the SUBCOM interface, or access TSO EXTERNAL or PERMANENT scope environment variables. The **envname** function returns **``OpenMVS''**, not **``TSO''**, for a process created by **oeattach**.

EXAMPLE

This example invokes the **ls** shell command as an MVS subtask, 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 in MVS batch or TSO). Use of **oeattach** rather than **fork** and **exec** may be a significant performance enhancement.

oeattach Create a Child Process as a Subtask

(continued)

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>
#include <lclib.h>

main()
{
   int pipefds[2];
   pid_t pid;
   char *const parmList[] = {"/bin/ls", "-l", "/u/userid/dirname",
                             NULL };
   char lsout[200];          /* buffer for out ls output         */
   int amt;
   int status;               /* status code from ls              */

   fclose(stdout);           /* avoid stdio interfering with fd 1 */
   pipe(pipefds);            /* create both ends of a pipe       */
   dup2(pipefds[1],STDOUT_FILENO);
                             /* make write end of pipe fd 1      */
   if (pipefds[1] != 1) close(pipefds[1]);
                             /* close write end                  */
   pid = oeattach("/bin/ls", parmList);
                             /* run ls command as subtask        */
   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));
}
```

RELATED FUNCTIONS

fork, **execv**, **oeattache**

oeattache Create a Child Process as a Subtask

SYNOPSIS

```
#include <sys/types.h>
#include <lclib.h>

pid_t oeattache(const char *file, char *const argv[],
                char *const envp[]);
```

DESCRIPTION

The **oeattche** function is identical to the **oeattach** function with one exception: it has an additional argument. The **envp** argument is a pointer to an array of pointers to null-terminated strings, which define the environment variables for the new process. A **NULL** pointer is used to mark the end of the array.

RETURN VALUE

If **oeattache** is successful, it returns the process ID of the new process. If it is unsuccessful, it returns **-1**.

CAUTIONS

Refer to the **oeattach**.

RELATED FUNCTIONS

fork, **execv**, **oeattach**

__passwd Verify or Change a Password

SYNOPSIS

```
#include <sys/types.h>
#include <pwd.h>

int __passwd(const char *userID, const char *currentPassword,
             const char *newPassword);
```

DESCRIPTION

__passwd verifies or changes a user password.

userID
identifies the user. **userID** is a 1 to 8 character, null-terminated, MVS user ID.

currentPassword
is the current password.

newPassword
is the new password. The new password replaces the current password. If **newPassword** is **NULL**, then **currentPassword** is verified and not changed.

The passwords are 1 to 8 character, null-terminated strings. Site-dependent restrictions on passwords may apply to both the **currentPassword** and **newPassword** arguments.

Note: This function can only be used by a process that is a member of the **BPX.DAEMON** class. See IBM's *Planning OpenEdition MVS* for more information.

Refer to IBM's *OpenEdition MVS Supplement for rlogin* (SC23-3847).

RETURN VALUE

__passwd returns a **0** if successful and a **-1** if unsuccessful. It is possible for **__passwd** to fail even if the password is correct, if the password has expired, and a new password was not specified.

pathconf Determine Pathname Variables that Can Be Configured

SYNOPSIS

```
#include <unistd.h>

long pathconf(const char *pathname, int configvar);
```

DESCRIPTION

pathconf determines the value of a configuration variable that is associated with a file or directory name. **pathname** is the file or directory. **configvar** is the configuration variable.

configvar is specified as one of the following symbols defined in **<unisted.h>**:

_PC_LINK_MAX	returns the maximum number of links possible for the file. If a directory is specified, **pathconf** returns the maximum number of links possible for the directory.
_PC_MAX_CANON	returns the maximum number of bytes in a terminal canonical input file. **pathname** refers to a character special file for a terminal.
_PC_MAX_INPUT	returns the minimum number of bytes for which space is available in a terminal input queue. This number is also the maximum number of bytes that a user can enter before a portable application reads the input. **pathname** refers to a character special file for a terminal.
_PC_NAME_MAX	returns the maximum number of characters in a filename. This number does not include a terminating NULL for filenames stored as strings. This limit is for names that do not specify any directories.
_PC_PATH_MAX	returns the maximum number of characters in a complete pathname. This number does not include a terminating NULL for pathnames stored as strings.
_PC_PIPE_BUF	returns the maximum number of bytes that can be written atomically to a pipe. If this number is exceeded, the operation can use more than one physical read and write operation to write and read the data. For FIFO special files, **pathconf** returns the value for the file itself. For directories, **pathconf** returns the value for FIFO special files that exist or can be created under the specified directory. For other kinds of files, an **errno** of **EINVAL** is stored.
_PC_CHOWN_RESTRICTED	returns whether or not the use of **chown** is restricted. For directories, **pathconf** returns the value for files, but not for subdirectories.

pathconf Determine Pathname Variables that Can Be Configured
(continued)

_PC_NO_TRUNC	returns whether or not a filename that is larger than the maximum name size is considered invalid, or is truncated. For directories, this applies to all files in the directory.
_PC_VDISABLE	returns the character, if any, which can be used to disable special character functions for a terminal file. **pathname** must refer to a character special file for the terminal.

RETURN VALUE

If successful, **pathconf** returns the value defined above for the particular value of **configvar**. If not successful, **pathconf** returns **-1**.

EXAMPLE

The following code fragment illustrates the use of **pathconf** to determine the maximum number of characters in a filename:

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

.
.
.
long result;

errno = 0;
if ((result = pathconf("/",_PC_NAME_MAX)) == -1)
   if (errno == 0)
      puts("NAME_MAX has no limit.");
   else perror("pathconf() error");
else
   print("NAME_MAX is %ld\n", result);
.
.
.
```

RELATED FUNCTIONS

fpathconf, **sysconf**

setegid Specify Effective Group ID

SYNOPSIS

```
#include <sys/types.h>

int setegid(gid_t groupID);
```

DESCRIPTION

setegid changes the effective group ID of the current process to the ID specified by **groupID**.

Note: This function can only be called by a superuser.

RETURN VALUE

setegid returns a **0** if successful and a **-1** if unsuccessful.

PORTABILITY

The **setegid** function is defined by the POSIX.1a standard.

RELATED FUNCTIONS

getegid, **getgid**, **setgid**, **setuid**

seteuid Specify Effective User ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int seteuid(uid_t userID);
```

DESCRIPTION

seteuid changes the effective user ID of the current process to the ID specified by **userID**.

Note: This function can only be called by a superuser.

RETURN VALUE

seteuid returns a `0` if successful and a `-1` if not successful.

PORTABILITY

The **seteuid** function is defined by the POSIX.1a standard.

RELATED FUNCTIONS

geteuid, **getuid**, **seteuid**, **setgid**

setgid Specify Group ID


SYNOPSIS

```
#include <sys/types.h>

int setgid(gid_t groupID);
```

DESCRIPTION

setgid sets one or more of the group IDs of the current process. **groupID** is the new group ID. **setgid** succeeds and sets the effective group ID to the real group ID if **groupID** is the same as the real group ID of the process. If **groupID** is not the same as the real group ID of the process, **setgid** succeeds only if the process belongs to a superuser, in which case it sets the real group ID, effective group ID, and saved set group ID to **groupID**.

RETURN VALUE

setgid returns a **0** if successful and a **−1** if unsuccessful.

EXAMPLE

The following example illustrates the use of **setgid** to set the effective group ID to the real group ID:

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

main()
{
   uid_t realGID, effectiveGID;

   realGID = getgid();
   effectiveGID = getegid();

   printf("Real group ID = %d\n", (int) realGID);
   printf("Effective group ID = %d\n", (int) effectiveGID);

   if (realGID != effectiveGID) {
      if (setgid(realGID) != 0)
         perror("setgid error");
      else
         printf("Effective group ID changed to %d.\n", (int) realGID);
   }
}
```

RELATED FUNCTIONS

getegid, **getgid**, **setegid**, **setuid**

setgroups Set Supplementary Group IDs

SYNOPSIS

```
#include <sys/types.h>
#include <grp.h>

int setgroups(int listSize, const gid_t groupList[]);
```

DESCRIPTION

setgroups sets the supplementary group ID list for the calling process to the list provided by the **groupList** argument. **listSize** specifies the size of the **groupList** array. The size of **groupList** cannot exceed the **NGROUPS_MAX** value, which is defined in **<grp.h>**. The **sysconf** function can be used to determine the value of **NGROUPS_MAX**.

Note: This function can only be called by a superuser.

RETURN VALUE

setgroups returns a **0** if successful and a **-1** if unsuccessful.

RELATED FUNCTIONS

getgroups, **initgroups**

setpgid Specify Process Group ID for Job Control

SYNOPSIS

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

DESCRIPTION

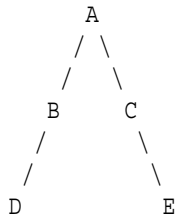
setpgid is used either to join an existing process group or to create a new process group. The process specified by **pid** (or the calling process if **pid** is **0**) is joined to the process group whose ID is **pgid**. If **pgid** is not an existing process group ID, then **pgid** must equal **pid** or be **0**, in which case a new process group with ID **pid** is created.

RETURN VALUE

setpgid returns a **0** if successful and a **−1** if not successful.

EXAMPLE

This example creates a number of processes, to illustrate the behavior of process groups and sessions. The process tree looks like:



Process D uses **setsid** to define itself as a new session. The other processes remain in the session of A. The processes A and B comprise one process group, while C and E comprise another. The **tcsetpgrp** function is used to cause C's process group to become the foreground process group.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

char process = 'A';       /* the process ID of the running process */

void status(void) {
     /* display interesting process IDs                              */
   printf("process %c: pid=%d, ppid=%d, pgid=%d, fg pgid=%d\n",
           process, (int) getpid(), (int) getppid(),
           (int) getpgrp(), (int) tcgetpgrp(STDIN_FILENO));
}
```

setpgid Specify Process Group ID for Job Control
(continued)

```
void erreport(char *func) {  /* report an error and abort          */
   fprintf(stderr, "process %c ", process);
   perror(func);
   abort();
}

void timeout(void) {         /* wait up to 10 seconds, then quit   */
   unsigned waitfor = 20;

   while(waitfor)
      waitfor = sleep(waitfor);
        /* wait quietly ignoring signals for 20 seconds            */
      printf("process %c: Terminating after 20 seconds.\n", process);
      exit(0);
}

void ttinhdlr(int signum) {
   printf("process %c: SIGTTIN detected.\n", process);
}

void usr1hdlr(int signum) {
  /* null handler - allows parent to wait for init of child        */
}

void termhdlr(int signum) {
   printf("process %c: SIGTERM detected, terminating.\n", process);
   exit(0);
}

void catchsig(void) {
   struct sigaction action;
   action.sa_handler = &ttinhdlr;
   sigemptyset(&action.sa_mask);
   action.sa_flags = 0;
   if (sigaction(SIGTTIN, &action, 0)) erreport("sigaction");
   action.sa_handler = &termhdlr;
   if (sigaction(SIGTERM, &action, 0)) erreport("sigaction");
   action.sa_handler = &usr1hdlr;
   if (sigaction(SIGUSR1, &action, 0)) erreport("sigaction");
}
```

setpgid Specify Process Group ID for Job Control
(continued)

```
pid_t createE(void) {
     /* fork and run process E - called by process C              */
   pid_t child;

   if ((child = fork()) != 0) return child;
     /* if running in parent, return child pid                    */
   process = 'E';
   catchsig();
   status();
   kill(getppid(), SIGUSR1); /* inform parent child is ready      */
   timeout();
   return -1;                /* should not occur                  */
}

pid_t createC(void) {
     /* fork and run process C - called by process A              */
   pid_t child;
   sigset_t newmask, oldmask;

   if ((child = fork()) != 0) return child;
     /* if running in parent, return child pid                    */
   process = 'C';
   catchsig();
   if (setpgid(0, 0) < 0)    /* create new process group         */
      erreport("setpgid");
   if (tcsetpgrp(STDIN_FILENO, getpid()))
                             /* become forgeground process group  */
      erreport("tcsetpgrp");
   status();
   if (createE() < 0) erreport("fork");
   pause();                  /* wait for SIGUSR1 from child       */
   sigemptyset(&newmask);
   sigaddset(&newmask, SIGTERM);
   sigprocmask(SIG_BLOCK, &newmask, &oldmask);
                             /* block SIGTERM until input read    */
   kill(getppid(), SIGUSR1); /* inform parent child is ready      */
   (void) getc(stdin);    /* attempt to read from stdin (terminal) */
   sigprocmask(SIG_SETMASK, &oldmask, 0);
                             /* allow SIGTERM to interrupt now    */
   timeout();
   return -1;                /* should not occur                  */
}

pid_t createD(void) {
     /* fork and run process D - called by process B              */
   pid_t child;

   if ((child = fork()) != 0) return child;
        /* if running in parent, return child pid                 */
   process = 'D';
   catchsig();
```

setpgid Specify Process Group ID for Job Control
(continued)

```
   setsid();                /* start new session                */
   status();
   kill(getppid(), SIGUSR1); /* inform parent child is ready      */
   timeout();
   return -1;               /* should not occur                 */
}

pid_t createB(void) {
     /* fork and run process B - called by process A             */
   pid_t child;

   if ((child = fork()) != 0) return child;
      /* if running in parent, return child pid                  */
   process = 'B';
   catchsig();
   status();
   if (createD() < 0) erreport("fork");
   pause();                 /* wait for SIGUSR1 from child      */
   kill(getppid(), SIGUSR1); /* inform parent child is ready      */
   timeout();
   return -1;               /* should not occur                 */
}

int main(void) {
   pid_t Cpid;              /* process id for C                 */
   pid_t Apid;              /* process id for A                 */

   Apid = getpid();
   status();
   printf("Enter carriage return to satisfy terminal read.\n");
   catchsig();
   if (createB() < 0) erreport("fork");
   pause();                 /* wait for SIGUSR1 from child      */
   if ((Cpid = createC()) < 0) erreport("fork");
   pause();                 /* wait for SIGUSR1 from child      */
   while (tcgetpgrp(STDIN_FILENO) == getpid())
      sleep(5);     /* wait for foreground process group to change */
   (void) getc(stdin);  /* attempt to read from stdin (terminal)   */
                        /* should generate SIGTTIN                 */
   sleep(2);            /* allow other processes to do their thing */
   kill(-Cpid, SIGTERM);/* terminate processes C and E             */
   kill(0, SIGTERM);    /* terminate processes A and B             */
                        /* D will remain for some seconds          */
}
```

RELATED FUNCTIONS

getpgrp, **setsid**, **tcsetpgrp**

setsid Create Session and Specify Process Group ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

DESCRIPTION

setsid creates a new session. The calling process is its session leader. The calling process is the process group leader of the new process group; it must not already be a process group leader. The calling process does not have a controlling terminal. The calling process begins as the only process in the new process group and in the new session. The process group ID of the new process group is equal to the process ID of the caller.

RETURN VALUE

setsid returns the value of the new process group ID of the calling process. **setsid** returns a **-1** if not successful.

EXAMPLE

The following code fragment illustrates the use of **setsid** to create a new session:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

pid_t pid;

.
.
.
printf("The process group ID is %d\n", (int) getpgrp());

if ((pid = setsid()) == -1)
   perror("setsid error");
else
   printf("The new process group ID is %d\n", (int) pid);
.
.
.
```

Note: Also see the **setpgid** example.

RELATED FUNCTIONS

setpgid

setuid Specify User ID

SYNOPSIS

```
#include <sys/types.h>
#include <unistd.h>

int setuid(uid_t userID);
```

DESCRIPTION

setuid changes the effective user ID (and possibly the real user ID) of the current process to the ID specified by **userID**. If **uid** is the process' real user ID, **setuid** sets the process' effective user ID. If **userID** is not the process' real user ID, **setuid** is allowed to proceed only if the calling process is a superuser process, in which case it sets all of the real user ID, effective user ID and saved set user ID as specified.

RETURN VALUE

setuid returns a **0** if successful and a **-1** if not successful.

EXAMPLE

The following example illustrates the use of **setuid** to set the effective user ID to the real user ID:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>

main()
{
   uid_t realUID, effectiveUID;

   realUID = getuid();
   effectiveUID = geteuid();

   printf("Real user ID = %d\n", (int) realUID);
   printf("Effective user ID = %d\n", (int) effectiveUID);

   if (realUID != effectiveUID) {
      if (setuid(realUID) != 0)
         perror("setuid() error");
      else
         printf("Effective user ID changed to %d.\n", (int) realUID);
   }
}
```

RELATED FUNCTIONS

geteuid, **getuid**, **seteuid**, **setgid**

sysconf Determine System Configuration Options

SYNOPSIS

```
#include <unistd.h>

long sysconf(int configOpt);
```

DESCRIPTION

sysconf returns the value of a system configuration option. The **configOpt** argument can be any of the following symbols, which specify the system configuration option to determine.

_SC_ARG_MAX
returns the value of **ARG_MAX**, which specifies the maximum number of bytes of argument and environment data that can be passed to an **exec** function.

_SC_CHILD_MAX
returns the value of **CHILD_MAX**, which specifies the maximum number of child processes that a user ID (UID) can have running at the same time.

_SC_CLK_TCK
returns the value of the **CLK_TCK** macro, which is defined in **<time.h>**. This macro specifies the number of clock ticks in a second.

_SC_JOB_CONTROL
returns the value of the **_POSIX_JOB_CONTROL** macro, which may be defined in **<unistd.h>**. This macro specifies that certain job control functions are implemented by the operating system. Certain functions, such as **setpgid**, will have more functionality if **_POSIX_JOB_CONTROL** is defined.

_SC_NGROUPS_MAX
returns the value of **NGROUPS_MAX**, which specifies the maximum number of supplementary group IDs (GIDs) that can be associated with a process.

_SC_OPEN_MAX
returns the value of **OPEN_MAX**, which specifies the maximum number of files that a single process can have open at one time.

_SC_SAVED_IDS
returns the value of the **SAVED_IDS** macro, which may be defined in **<unistd.h>**. This macro specifies that a saved set UID and a saved set GID are implemented. The behavior of certain functions, such as **setuid** and **setgid**, is affected by this macro.

_SC_STREAM_MAX
returns the value of the **STREAM_MAX** macro, which may be defined in **<unistd.h>**. This macro indicates the maximum number of streams that a process can have open simultaneously.

_SC_TZNAME_MAX
returns the value of the **TZNAME_MAX** macro, which may be defined in **<unistd.h>**. This macro indicates the maximum length of the name of a time zone.

sysconf Determine System Configuration Options
(continued)

_SC_VERSION
returns the value of the **VERSION** macro, which may be defined in **<unistd.h>**. This macro indicates the version of the POSIX.1 standard that the system conforms to.

RETURN VALUE

sysconf returns the value associated with the option specified by the **configOpt** argument. If the symbol corresponding to the specified option exists but is not supported, **sysconf** returns **−1** but does not change the value of **errno**. If **sysconf** fails in any other way, it returns **−1** and sets **errno** to **EINVAL**.

EXAMPLE

The following example illustrates the use of **sysconf** to obtain the value of **STREAM_MAX**:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

main()
{
   int max_streams;

   errno = 0;

   if ((max_streams = sysconf(_SC_STREAM_MAX)) == -1)
      if (errno == 0)
         printf("STREAM_MAX not supported by this implementation.\n");
      else
         perror("sysconf error.");
   else
      printf("STREAM_MAX = %d\n", max_streams);
}
```

RELATED FUNCTIONS

fpathconf, **pathconf**

tcdrain Wait for Output to Drain

SYNOPSIS

```
#include <termios.h>

int tcdrain(int fileDescriptor);
```

DESCRIPTION

tcdrain blocks the calling process until all output sent to the terminal device referred to by the file descripter **fileDescriptor** has been sent. If **fileDescriptor** refers to the controlling terminal, this function cannot be called from a background process unless the **SIGTTOU** signal is blocked. By default, **tcdrain** will return a **-1** (unsuccessful) if **SIGTTOU** is generated.

Note that all terminal output is considered to be sent when it has been transmitted to the OMVS command, even if all the data have not yet been actually displayed.

RETURN VALUE

tcdrain returns a **0** if successful and a **-1** if unsuccessful.

EXAMPLE

The following example illustrates the use of **tcdrain** to block a calling process until all output is sent to a terminal, and then to write an output line to that terminal:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int ttyDevice = STDOUT_FILENO;
   char * lineOut = "Jack be nimble!";

      /* Make sure file descriptor is for a TTY device.           */
   if ( ! isatty(ttyDevice) ) {
      printf("Not a TTY device.\n");
      return(EXIT_FAILURE);
   }
```

tcdrain Wait for Output to Drain
(continued)

```
            /* Wait for all data transmission to the terminal to finish  */
            /* and then write a line to the terminal.                    */
         else {
            if (tcdrain(ttyDevice) != 0)
               perror("tcdrain error");
            else
               write(ttyDevice, lineOut, strlen(lineOut));
         }
         return(EXIT_SUCCESS);
      }
```

RELATED FUNCTIONS

tcflow, **tcflush**

tcflow Controls the Flow of Data to a Terminal

SYNOPSIS

```
#include <termios.h>

int tcflow(int fileDescriptor, int action);
```

DESCRIPTION

tcflow controls the flow of data to and from a terminal device. Because OpenEdition MVS supports only pseudo-terminals, **tcflow** affects only the flow of data between the OMVS TSO command and user programs. It does not directly affect transmission to and from a user's 3270 terminal.

The arguments to **tcflow** are:

fileDescriptor
is a file descriptor that refers to a terminal device.

action
is a symbolic constant that controls the action of **tcflow**. The values for **action** are defined in **<termios.h>** and can be any of the following:

TCOOFF	suspends output to the terminal.
TCOON	restarts suspended output to a terminal.
TCIOFF	transmits a **STOP** character to a terminal. This should stop further transmission of data from the terminal.
TCION	transmits a **START** character to a terminal. This should restart the transmission of data from the terminal.

If **tcflow** is called from a background process, with a file descriptor that refers to the controlling terminal for the process, a **SIGTTOU** signal may be generated. This will cause the function call to be unsuccessful, returning a **-1**. If **SIGTTOU** is blocked, the function call proceeds normally.

RETURN VALUE

tcflow returns a **0** if successful and a **-1** if unsuccessful.

EXAMPLE

The following example illustrates the use of **tcflow** to suspend data transmission:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int ttyDevice = STDOUT_FILENO;

      /* Make sure file descriptor is for a TTY device,          */
   if ( ! isatty(ttyDevice) ) {
```

tcflow Controls the Flow of Data to a Terminal
(continued)

```
        printf("Not a TTY device.\n");
        return(EXIT_FAILURE);
    }

        /* Suspend and resume data transmission,                    */
    else {
        printf("About to suspend data transmission for 5 seconds.\n");
        if (tcflow(ttyDevice, TCOOFF) != 0) {
            perror("tcflow error");
            exit(EXIT_FAILURE);
        }
        sleep(5);
        if (tcflow(ttyDevice, TCOON) == 0)
            printf("Data transmisiion suspended and resumed.\n");
        else { /* We turned it off, and now we can't turn it back on!  */
            perror("tcflow error");  /* probably message will be lost   */
            abort();
    }
    return(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

tcdrain, **tcflush**

tcflush Flush Terminal Input or Output

SYNOPSIS

```
#include <termios.h>

int tcflush(int fileDescriptor, int queue);
```

DESCRIPTION

tcflush is called by a process to flush all input that has received but not yet been read by a terminal, or all output written but not transmitted to the terminal. Flushed data are discarded and cannot be retrieved.

fileDescriptor
is a file discriptor that refers to a terminal device.

queue
is a symbolic constant that specifies which queue to flush. The values for **queue** are defined in **<termios.h>** and can be any of the following:

TCIFLUSH	flushes the input queue, which contains data that have been received but not yet read.
TCOFLUSH	flushes the output queue, which contains data that have been written but not yet transmitted.
TCIOFLUSH	flushes both the input and output queue.

RETURN VALUE

tcflush returns a **0** if successful and a **-1** if unsuccessful. If **tcflush** is called from a background process with a file descriptor that refers to the controlling terminal for the process, a **SIGTTOU** signal may be generated. This will cause the function call to be unsuccessful, returning a **-1** and setting **errno** to **EINTR**. If **SIGTTOU** is blocked, the function call proceeds normally.

EXAMPLE

The following example illustrates the use of **tcflush** to flush both the input and output queues of a terminal:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int ttyDevice = STDOUT_FILENO;

      /* Make sure file descriptor is for a TTY device.   */
   if ( ! isatty(ttyDevice) ) {
      printf("Not a TTY device.\n");
      return(EXIT_FAILURE);
   }
```

tcflush Flush Terminal Input or Output
(continued)

```
        /* Flush both the input and output queues.          */
    else {
        if (tcflush(ttyDevice, TCIOFLUSH) == 0)
            printf("The input and output queues have been flushed.\n");
        else
            perror("tcflush error");
    }
    return(EXIT_SUCCESS);
}
```

RELATED FUNCTIONS

tcdrain, **tcflow**

tcgetattr Get Terminal Attributes

SYNOPSIS

```
#include <termios.h>

int tcgetattr(int fileDescriptor, struct termios *terminal);
```

DESCRIPTION

tcgetattr stores the attributes of a terminal device in a **termios** structure. The fields of **termios** (declared in **<termios.h>**) are flags that identify terminal modes and control characters. The following arguments are passed to **tcgetattr**:

fileDescriptor
is a file discriptor that refers to a terminal device.

terminal
is the address of a **termios** structure. The **tcgetattr** function fills this structure with the attributes of the terminal referred to by **fileDescriptor**.

You can call **tcgetattr** from either a foreground or background process. However, if called from a background process, terminal attributes may be changed by a subsequent call from a foreground process.

RETURN VALUE

tcgetattr returns a **0** if successful and a **-1** if unsuccessful.

EXAMPLE

The following example illustrates the use of **tcgetattr** to determine the terminal attributes for **stdout**:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define NOTATTY 1

main()
{
   int ttyDevice = STDOUT_FILENO;
   struct termios termAttributes;

      /* Make sure file descriptor is for a TTY device.         */
   if ( ! isatty(ttyDevice) )
      return(NOTATTY);
```

tcgetattr Get Terminal Attributes
(continued)

```
            /* Get terminal attributes and then determine if terminal */
            /* start and stop is enabled and if terminal is in        */
            /* canonical mode.                                        */
         else {
            if (tcgetattr(ttyDevice, &termAttributes) != 0)
               perror("tcgetattr error");
            else {
               if (termAttributes.c_iflag & IXON)
                   printf("Terminal start and stop is enabled.\n");
               if (termAttributes.c_lflag & ICANON)
                   printf("Terminal is in canonical mode.\n");
            }
         }
         return(EXIT_SUCCESS);
      }
```

RELATED FUNCTIONS

cfgetispeed, **cfgetospeed**, **tcsetattr**

tcgetpgrp Get Foreground Process Group Identification

SYNOPSIS

```
#include <unistd.h>

int tcgetpgrp(int fileDescriptor);
```

DESCRIPTION

tcgetpgrp returns the process group identification (PGID) for the foreground process group associated with a controlling terminal that is referred to by the file descriptor **fileDescriptor**.

The **tcgetpgrp** function can be called from a background process to obtain the PGID of a foreground process group. The process can then change from the background group to the foreground process group by making a call to **tcsetpgrp**, specifying the PGID of the new foreground group as one of the arguments passed to **tcsetpgrp**.

RETURN VALUE

If successful, **tcgetpgrp** returns the value of the forground process group identification (PGID). A **-1** is returned if unsuccessful.

EXAMPLE

The following example illustrates the use of **tcgetpgrp** to obtain the PGIDs for **stdin**, **stdout**, and **stderr**:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   pid_t stdin_PGID, stdout_PGID, stderr_PGID;

      /* Get the PGIDs for stdin, stdout, and stderr.  */
   stdin_PGID = tcgetpgrp(STDIN_FILENO);
   if (stdin_PGID == -1) {
      printf("Could not get PGID for stdin.\n");
      return(EXIT_FAILURE);
   }
   stdout_PGID = tcgetpgrp(STDOUT_FILENO);
   if (stdout_PGID == -1) {
      printf("Could not get PGID for stdout.\n");
      return(EXIT_FAILURE);
   }
   stderr_PGID = tcgetpgrp(STDERR_FILENO);
   if (stderr_PGID == -1) {
      printf("Could not get PGID for stderr.\n");
      return(EXIT_FAILURE);
```

tcgetpgrp Get Foreground Process Group Identification
(continued)

```
    }

    printf("The PGID for stdin is %d.\n", stdin_PGID);
    printf("The PGID for stdout is %d.\n", stdout_PGID);
    printf("The PGID for stderr is %d.\n", stderr_PGID);

    return(EXIT_SUCCESS);
}
```

Note: Also see the **setpgid** example.

RELATED FUNCTIONS

getpgrp, **tcsetpgrp**

tcsendbreak Send a Break Condition

SYNOPSIS

```
#include <termios.h>

int tcsendbreak(int fileDescriptor, int duration);
```

DESCRIPTION

tcsendbreak sends a break condition to a terminal.

fileDescriptor
is a file discriptor that refers to an asynchronous communications line.

duration
controls the duration of the break condition in an implementation-defined way. See the POSIX.1 standard for details.

Under MVS OpenEdition, all terminals are pseudoterminals. The **tcsendbreak** function has no effect on pseudoterminals.

RETURN VALUE

tcsendbreak returns a **0** if successful and a **-1** if unsuccessful. If **tcsendbreak** is called from a background process, with a file descriptor that refers to the controlling terminal for the process, a **SIGTTOU** signal may be generated. This will cause the function call to be unsuccessful, returning a **-1** and setting **errno** to **EINTR**. If **SIGTTOU** is blocked, the function call proceeds normally.

EXAMPLE

The following example illustrates the use of **tcsendbreak** to transmit a break condition to **stdout**:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int ttyDevice = STDOUT_FILENO;
   int time = 15;
   char * lineOut = "Break transmitted to terminal.";

      /* Wait for all data transmission to the terminal to finish */
      /* and then transmit a break condition to the terminal.     */
   if (tcdrain(ttyDevice) != 0) {
      perror("tcdrain error");
      return(EXIT_FAILURE);
   }
```

tcsendbreak Send a Break Condition

(continued)

```
        else {
           if (tcsendbreak(STDOUT_FILENO, time) != 0) {
              perror("tcdsendbreak error");
              return(EXIT_FAILURE);
           }
           else
              write(ttyDevice, lineOut, strlen(lineOut) + 1);
        }
        return(EXIT_SUCCESS);
     }
```

RELATED FUNCTIONS

tcflow

tcsetattr Set Terminal Attributes

SYNOPSIS

```
#include <termios.h>

int tcsetattr(int fileDescriptor, int actions,
              struct termios *terminto);
```

DESCRIPTION

tcsetattr set the attributes of a terminal device. The attributes are stored in a **termios** structure prior to calling the **tcsetattr** function. The normal sequence is to call **tcgetattr** to obtain the current attribute settings, modify the **termios** structure to contain the new settings, and then set the terminal attributes with a call to **tcsetattr**.

The following arguments are passed to **tcsetattr**:

fileDescriptor
is a file discriptor that refers to a terminal device.

action
is a symbolic constant that controls when the attributes are set. The values of **action** are defined in **<termios.h>** and may be one of the following:

TCSANOW	sets terminal attributes immediately.
TCSADRAIN	sets the terminal attributes after all output that has aleady been written to the terminal is transmitted. **TCSADRAIN** should be specified when setting terminal attributes that could affect output.
TCSAFLUSH	sets the terminal attributes after all output that has already been written to the terminal is transmitted. Input that has been received, but not read, is discarded.

terminto
is the address of a **termios** structure. The members of **termios**, which is declared in **<termios.h>**, are flags that identify terminal modes and control characters. The **tcsetattr** function sets the attributes of the terminal referred to by **fileDescriptor** based on the information contained in this structure.

termios structure

The **termios** structure is declared in **<termios.h>** as follows:

```
typedef int tcflag_t;
typedef char cc_t;

#define NCCS 11          /* number of special control characters */
```

tcsetattr Set Terminal Attributes
(continued)

```
struct termios {
   tcflag_t c_cflag;     /* control modes                        */
   tcflag_t c_iflag;     /* input modes                          */
   tcflag_t c_lflag;     /* local modes                          */
   tcflag_t c_oflag;     /* output modes                         */
   cc_t c_cc[NCCS];      /* control characters and values        */
};
```

Each of the members of type **tcflag_t** is constructed from bitmasks that are also declared in **<termios.h>**. The **c_cc[NCCS]** array contains control characters that have special meaning for terminal handling.

Refer to *The POSIX.1 Standard: A Programmer's Guide*, by Fred Zlotnick, for portable information about the bitmasks of the **termios** structure. Also refer to *Assembler Callable Services for OpenEdition MVS* (SC23-3020) for operating-system-specific details.

RETURN VALUE

tcsetattr returns a **0** if successful and a **-1** if unsuccessful. If **tcsetattr** is called from a background process, with a file descriptor that refers to the controlling terminal for the process, a **SIGTTOU** signal may be generated. This will cause the function call to be unsuccessful, returning a **-1** and setting **errno** to **EINTR**. If **SIGTTOU** is blocked, the function call proceeds normally.

EXAMPLE

The following example illustrates the use of **tcsetattr** to change the terminal attributes of a TTY device:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define NOTATTY 1

main()
{
   int ttyDevice = STDOUT_FILENO;
   struct termios termAttributes;

      /* Make sure file descriptor is for a TTY device.        */
   if ( ! isatty(ttyDevice) )
      return(NOTATTY);

      /* Get terminal attributes and then determine if terminal */
      /* start and stop is enabled. If IXON is on, turn it off  */
      /* and call tcsetattr.                                    */
```

tcsetattr Set Terminal Attributes
(continued)

```
        else {
           if (tcgetattr(ttyDevice, &termAttributes) != 0) {
              perror("tcgetattr error");
              return(EXIT_FAILURE);
           }
           else {
              if (termAttributes.c_iflag & IXON) {
                 termAttributes.c_iflag = termAttributes.c_iflag ^ IXON;
                 if (tcsetattr(ttyDevice, TCSANOW, &termAttributes) != 0) {
                    perror("tcsetattr error");
                    return(EXIT_FAILURE);
                 }
                 printf("IXON disabled. START and STOP characters \n");
                 printf("will be passed to the application.\n");
                 }
              else
                 printf("IXON was already set to 0.\n");
           }
        }
        return(EXIT_SUCCESS);
     }
```

RELATED FUNCTIONS

cfsetispeed, **cfsetospeed**,

tcsetpgrp Set Foreground Process Group Identification

SYNOPSIS

```
#include <unistd.h>

int tcsetpgrp(int fileDescriptor, pid_t processID);
```

DESCRIPTION

tcsetpgrp sets the process group identification (PGID) for the foreground process group associated with a controlling terminal. The arguments to **tcsetpgrp** are:

fileDescriptor
is a file discriptor that refers to a terminal device. The file descriptor must be for the controlling terminal associated with the process calling **tcsetpgrp**.

processID
is a foreground process group identification number (PGID) from the same session as the process calling **tcsetpgrp**.

The **tcgetpgrp** function can be called from a background process to obtain the PGID of a foreground process group. The process can then change from the background group to the foreground process group by making a call to **tcsetpgrp** and specifying the PGID of the new foreground group as one of the arguments passed to **tcsetpgrp**.

After the PGID for a terminal has been changed, reads by the process group that was associated with the terminal prior to the call to **tcsetpgrp** either fail or cause the generation of a **SIGTTIN** signal. A **SITTIN** signal causes the process group to stop. Depending upon the setting of the **TOSTOP** bit in the **termios** structure, writes may also cause the process group to stop due to the generation of a **SIGTTOU** signal. (Refer to the **tcsetattr** function for a description of **termios**.)

RETURN VALUE

If successful, **tcsetpgrp** returns a **0**, and a **−1** is returned if unsuccessful.

EXAMPLE

The following example illustrates the use of **tcsetpgrp** to set the PGID for **stdin**:

```
#include <sys/types.h>
#include <termios.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   pid_t stdin_PGID;
```

tcsetpgrp Set Foreground Process Group Identification

(continued)

```
      /* Get the PGIDs for stdin. */
   stdin_PGID = tcgetpgrp(STDIN_FILENO);
   if (stdin_PGID == -1) {
      printf("Could not get PGID for stdin.\n");
      return(EXIT_FAILURE);
   }
   else if (tcsetpgrp(STDIN_FILENO, stdin_PGID) == -1) {
      printf("Could not set PGID.\n");
      return(EXIT_FAILURE);
   }

   printf("The PGID has been changed to %d.\n", stdin_PGID);
   return(EXIT_SUCCESS);
}
```

Note: Also see the **setpgid** example.

RELATED FUNCTIONS

tcgetpgrp

times Determine Process Times


SYNOPSIS

```
#include <time.h>

clock_t times(struct tms *buffer);
```

DESCRIPTION

times stores user and system CPU time for a process and its children. The time information is stored in a **tms** structure located at the address pointed to by **buffer**. The **tms** structure contains the following members:

clock_t tms_utime
is the amount of CPU time used by instructions in the calling process. Under MVS OpenEdition, this does not include time used by instructions in the kernel; however, it does include CPU time for the address space before it became a process.

clock_t tms_stime
is the amount of CPU time used by the system on behalf of the calling process. Under MVS OpenEdition, this value represents kernel time used for the calling process. It does not include time spent calling other system functions on behalf of the calling process.

clock_t tms_cutime
is the total user time for all terminated child processes. This value is calculated by summing the **tms_utime** and **tms_cutime** values for all child processes that have terminated.

clock_t tms_cstime
is the total system time for all terminated child processes. This value is calculated by summing the **tms_stime** and **tms_cstime** values for all child processes that have terminated.

The **clock_t** type measures time in clock ticks. The amount of time represented by each clock tick is determined by the **CLK_TCK** symbol, which is defined in **time.h**.

RETURN VALUE

If successful, **times** returns a positive value representing elapsed time, and **((clock_t) -1)** if it is unsuccessful. The elapsed time value is referenced from an arbitrary point; hence, it is only useful when making more than one call to **times**.

Note: OpenEdition computes elapsed time values using fullword (long) arithmetic. If this computation overflows, **times** indicates an error and sets **errno** to **ERANGE**.

EXAMPLE

The following example illustrates the use of **times** to determine process times:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <times.h>
```

times Determine Process Times
(continued)

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

int main(void)
{
   int i, status;
   pid_t pid;
   time_t currentTime;
   struct tms cpuTime;

   if ((pid = fork()) == -1) {         /* Start a child process.   */
        perror("fork error");
        exit(EXIT_FAILURE);
   }
   else if (pid == 0) {                /* This is the child.       */
      time(&currentTime);
      printf("Child process started at %s", ctime(&currentTime));
      for (i = 0; i < 5; ++i) {
         printf("Counting: %d\n", i);  /* count for 5 seconds.     */
         sleep(1);
      }
      time(&currentTime);
      printf("Child process ended at %s", ctime(&currentTime));
      exit(EXIT_SUCCESS);
   }
   else {                              /* This is the parent.      */
      time(&currentTime);
      printf("Parent process started at %s", ctime(&currentTime));

      if (wait(&status) == -1 )        /* Wait for child process.  */
         perror("wait error");
      if (WIFEXITED(status))
         printf("Child process ended normally.\n");
      else
         printf("Child process did not end normally.\n");

      if (times(&cpuTime) < 0)         /* Get process times.       */
         perror("times error");
      else {
         printf("Parent process user time = %f\n",
              ((double) cpuTime.tms_utime)/CLK_TCK);
         printf("Parent process system time = %f\n",
              ((double) cpuTime.tms_stime)/CLK_TCK);
         printf("Child process user time = %f\n",
              ((double) cpuTime.tms_cutime)/CLK_TCK);
         printf("Child process system time = %f\n",
              ((double) cpuTime.tms_cstime)/CLK_TCK);
      }
```

times Determine Process Times
(continued)

```
            time(&currentTime);
            printf("Parent process ended at %s", ctime(&currentTime));
            exit(EXIT_SUCCESS);
        }
    }
```

RELATED FUNCTIONS

clock

umask Change File Mode Creation Mask


SYNOPSIS

```
#include <sys/stat.h>
#include <sys/types.h>

mode_t umask(mode_t creationMask);
```

DESCRIPTION

umask changes the file mode creation mask to the value specified by **creationMask**. The file mode creation mask controls which permission bits may be set when a file is created. Bits that are set to 1 in the file mode creation mask are used to mask the corresponding permission bits when a file is created. For example, setting the file group read permission bit, **S_IRGRP**, in the file mode creation mask will prevent this bit from being set when a new file is created.

The value of **creationMask** is formed by ORing any of the following symbols, which are defined in the **<stat.h>** include file:

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_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

umask always returns the previously defined file mode creation mask. There is no way to read the value of the creation mask without changing it. It requires two calls to **umask** to determine the value of the mask: first you read the original value, then you reset the mask to the original value.

umask Change File Mode Creation Mask
(continued)

EXAMPLE

The following code fragment illustrates the use of **umask** to change the file mode creation mask:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

main()
{
   mode_t oldMask, newMask;

      /* Get old mask, temporarily setting the mask to 0.            */
   oldMask = umask((mode_t) 0);

      /* Print old mask. Octal values are used by mask.              */
   printf("Old mask = %o\n", (int) oldMask);

      /* Make sure group read permission is allowed.                 */
   if (oldMask & S_IRGRP) {
      printf("Changing group read permission from MASKED to UNMASKED.\n");
      oldMask = (oldMask ^ S_IRGRP);
   }
      /* Make sure group write and execute permission is not allowed. */
   newMask = (oldMask|S_IWGRP|S_IXGRP);

   umask(newMask);                              /* Update mask.     */
   printf("New mask = %o\n\n", (int) newMask);  /* Print new mask.  */

   printf("The file mode creation mask now specifies:\n\n");
   printf("     Group read permission      UNMASKED\n");
   printf("     Group write permission     MASKED\n");
   printf("     Group execute permission   MASKED\n");
}
```

RELATED FUNCTIONS

ccreat, **mkdir**, **mkfifo**, **mknod**, **open**

umount Unmounts a File System

SYNOPSIS

```
#include <sys/stat.h>

int umount(const char *fileSystem, mtm_t mountMode);
```

DESCRIPTION

umount unmounts a file system from the hierarchical file system. The **fileSystem** argument is a pointer to a null-terminated string containing the name of the file system to be removed.

The name specified by **fileSystem** must match the name specified as an argument to the **mount** function when the file system was mounted. Like the **mount** function, **umount** can be issued only by a superuser.

mountMode can be any one of the following symbolic names:

MTM_UMOUNT
is a normal unmount request. The unmount is performed if the specified file system is not in use, and the request is rejected if it is in use.

MTM_DRAIN
is an unmount drain request. **MTM_DRAIN** waits until all use of the specified file system is terminated and then unmounts the file system.

MTM_IMMED
is an immediate unmount request. The specified file system is unmounted immediately. Any users of files in the specified file system undergo a forced failure. All data changes that were made prior to the immediate unmount request are saved. If the data cannot be saved, the immediate unmount request fails.

MTM_FORCE
is a forced unmount request. The specified file system is unmounted immediately. Any users of files in the specified file system undergo a forced failure. All data changes that were made prior to the immediate unmount request are saved. However, unlike the immediate unmount request, a forced unmount will continue even if the data cannot be saved. As a result, data may be lost with a forced unmount request.

MTM_RESET
is a reset unmount request. The reset request is used to stop an unmount drain request that is waiting for the file system to become available before performing the unmount operation.

PORTABILITY

The **umount** function is useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

RETURN VALUE

umount returns a `0` if successful and a `-1` if unsuccessful.

umount Unmounts a File System
(continued)

EXAMPLE

The following code fragment illustrates the use of **umount** to unmount a file system:

```
#include <sys/types.h>
#include <sys/stat.h>

main()
{
char *HFS = "POSIX.FILE.SYSTEM";
.
.
.
umount(HFS, MTM_DRAIN);
.
.
.
}
```

RELATED FUNCTIONS

mount, **w_getmntent**

uname Display Current Operating System Name


SYNOPSIS

```
#include <sys/utsname.h>

int uname(struct utsname *sysInfo);
```

DESCRIPTION

uname stores information about the operating system you are running under in a structure at the location pointed to by **sysInfo**. The **utsname** structure is declared in **<sys/utsname.h>** and has the following elements:

char *sysname;
points to the name of the operating system implementation.

char *nodename;
points to the node name within a communications network for the specific implementation.

char *release;
points to the current release level for the implementation.

char *version;
points to the current version number for the release.

char *machine;
points to the name of the machine on which the operating system is running.

RETURN VALUE

uname returns a nonnegative value if successful and a **-1** if unsuccessful.

EXAMPLE

The following example illustrates the use of **uname** to determine information about the operating system:

```
#include <sys/types.h>
#include <sys/utsname.h>
#include <stdio.h>

main()
{
   struct utsname sysInfo;

   if (uname(&sysInfo) != -1) {
      puts(sysInfo.sysname);
      puts(sysInfo.nodename);
      puts(sysInfo.release);
      puts(sysInfo.version);
      puts(sysInfo.machine);
   }
   else
      perror("uname() error");
}
```

uname Display Current Operating System Name
(continued)

RELATED FUNCTIONS

sysname, **syslevel**

wait Wait for Child Process to End


SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *statusPtr);
```

DESCRIPTION

wait suspends the calling process until one of its child processes ends. Usually, the **wait** function is called before a child process terminates, in which case the parent process waits for a child process to end; however, if the system already has information about a terminated child process when **wait** is called, the return from **wait** occurs immediately. **wait** also returns if a signal is received and is not ignored.

Status information for the terminating child process is usually stored at the location pointed to by **statusPtr**; however, there is one situation when status information is not available after a call to **wait**: if **wait** is called with **NULL** as the **statusPrt** value, no status information is returned. Assuming that this situation does not apply, the macros described in the next section are used to analyze the status information.

Analyzing Status Information

After the call to **wait**, status information stored at the location pointed to by **statusPtr** can be evaluated with the following macros:

WIFEXITED(*statusPtr)
evaluates to a nonzero (true) value if the child process terminates normally.

WEXITSTATUS(*statusPtr)
if the child process terminates normally, this macro evaluates to the lower 8 bits of the value passed to the **exit** or **_exit** function or returned from **main**.

WIFSIGNALED(*statusPtr)
evaluates to a nonzero (true) value if the child process terminates because of an unhandled signal.

WTERMSIG(*statusPtr)
if the child process ends by a signal that was not caught, this macro evaluates to the number of that signal.

RETURN VALUE

If successful, **wait** returns the process ID of the child process. If unsuccessful, a **-1** is returned.

CAUTIONS

The **wait** function will terminate if a signal managed by OpenEdition arrives before any child process has terminated. It will not terminate if a signal managed by SAS/C arrives; the signal will remain pending until the completion of **wait**.

Older UNIX programs may manipulate the status information stored at the location pointed to by **statusPtr** directly. These programs must be changed to use the POSIX.1 defined macros. This is necessary because OpenEdition and

wait Wait for Child Process to End
(continued)

SAS/C use different values for signal numbers, and the status codes reflect OpenEdition signal numbers, not SAS/C signal numbers. For instance, a process terminated with the **SIGKILL** signal will have status code 9, but the comparison (**status == SIGKILL**) will fail. A correct POSIX-conforming test for process termination by **SIGKILL** would be as follows:

```
(WIFSIGNALED(status) && WTERMSIG(status) == SIGKILL)
```

The **WTERMSIG** macro performs conversion of OpenEdition signal numbers to SAS/C signal numbers.

EXAMPLE

The following example illustrates the use of **wait** to suspend a parent process until a child terminates:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

main()
{
   int status, randomNumber, seed;
   char buffer[80];
   pid_t pid;

   if ((pid = fork()) == -1) {
        perror("fork error");
        exit(EXIT_FAILURE);
   }

   else if (pid == 0) {             /* start of child process      */
      printf("Child process started.\n");
      printf("Enter a random seed.\n");
      if(gets(buffer)){
         seed = atoi(buffer);
         srand(seed);
      }
      randomNumber = rand();        /* random end to child process */
      if ((randomNumber % 2) == 0) {
         printf("Child process aborted; pid = %d.\n", (int) pid);
         abort();
         fclose(stdout);
      }
      else {
         printf("Child process ended normally; pid = %d.\n", (int) pid);
         exit(EXIT_SUCCESS);
      }
   }
```

wait Wait for Child Process to End
(continued)

```
   else {                          /* start of parent process     */
      printf("Parent process started.\n");
      if ((pid = wait(&status)) == -1)
                                   /* Wait for child process.     */
         perror("wait error");
      else {                       /* Check status.               */
         if (WIFSIGNALED(status) != 0)
            printf("Child process ended because of signal %d.\n",
                    WTERMSIG(status));
         else if (WIFEXITED(status) != 0)
            printf("Child process ended normally; status = %d.\n",
                    WEXITSTATUS(status));
         else
            printf("Child process did not end normally.\n");
      }
      printf("Parent process ended.\n");
      exit(EXIT_SUCCESS);
   }
}
```

RELATED FUNCTIONS

fork, **waitpid**

waitpid Wait for a Specific Process to End

SYNOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statusPtr, int options);
```

DESCRIPTION

waitpid suspends the calling process until a specified process terminates. When the specified process ends, status information from the terminating process is stored in the location pointed to by **statusPtr** and the calling process resumes execution. If the specified process has already ended when the **waitpid** function is called and the system has status information, the return from **waitpid** occurs immediately. A return from the **waitpid** function also occurs if a signal is received and is not ignored.

pid
specifies a process, normally a child process, that the calling process waits for. The process is identified as follows:

- If **pid** is greater than 0, the calling process waits for the process whose process identification number (PID) is equal to the value specified by **pid**.
- If **pid** is equal to 0, the calling process waits for the process whose process group ID is equal to the PID of the calling process.
- If **pid** is equal to -1, the calling process waits for any child process to terminate.
- If **pid** is less than -1, the calling process waits for the process whose process group ID is equal to the absolute value of **pid**.

statusPtr
is a pointer to the location where status information for the terminating process is to be stored. There is one situation when status information is not available after a call to **waitpid**: if **waitpid** is called with **NULL** as the **statusPrt** value, no status information is returned. Assuming that this situation does not apply, the macros described in "Analyzing Status Information" on page 20-111 are used to analyze the status information.

options
specifies optional actions for the **waitpid** function. Either of the following option flags may be specified, or they can be combined with a bitwise inclusive OR operator:

WNOHANG
causes the call to **waitpid** to return status information immediately, without waiting for the specified process to terminate. Normally, a call to **waitpid** causes the calling process to be blocked until status information from the specified process is available; the **WNOHANG** option prevents the calling process from being blocked. If status information is not available, **waitpid** returns a **0**.

waitpid Wait for a Specific Process to End
(continued)

WUNTRACED
causes the call to **waitpid** to return status information for a specified process that has either stopped or terminated. Normally, status information is returned only for terminated processes.

Analyzing Status Information

After the call to **waitpid**, status information stored at the location pointed to by **statusPtr** can be evaluated with the following macros:

WIFEXITED(*statusPtr)
evaluates to a nonzero (true) value if the specified process terminated normally.

WEXITSTATUS(*statusPtr)
if the specified process terminated normally, this macro evaluates the lower 8 bits of the value passed to the **exit** or **_exit** function or returned from **main**.

WIFSIGNALED(*statusPtr)
evaluates to a nonzero (true) value if the specified process terminated because of an unhandled signal.

WTERMSIG(*statusPtr)
if the specified process is ended by an unhandled signal, this macro evaluates to the number of that signal.

WIFSTIPPED(*statusPtr)
evaluates to a nonzero (true) value if the specified process is currently stopped but not terminated.

WSTOPSIG(*statusPtr)
if the specified process is currently stopped but not terminated, then this macro evaluates to the number of the signal that caused the process to stop

RETURN VALUE

If successful, **waitpid** returns the process ID of the terminated process whose status was reported. If unsuccessful, a **-1** is returned.

CAUTIONS

If **W_NOHANG** is not specified, the **waitpid** function will terminate if a signal managed by OpenEdition arrives before any child process has terminated. It will not terminate if a signal managed by SAS/C arrives; the signal will remain pending until the completion of **waitpid**.

PORTABILITY

The **waitpid** function is defined by the POSIX.1 standard.

Older UNIX programs may manipulate the status information stored at the location pointed to by **statusPtr** directly. These programs must be changed to use the POSIX.1 defined macros. This is necessary because OpenEdition and SAS/C use different values for signal numbers, and the status codes reflect OpenEdition signal numbers, not SAS/C signal numbers. For instance, a process terminated with the **SIGKILL** signal will have status code **9**, but the

waitpid Wait for a Specific Process to End
(continued)

comparison (**status == SIGKILL**) will fail. A correct POSIX-conforming test for process termination by **SIGKILL** would be as follows:

```
(WIFSIGNALED(status) && WTERMSIG(status) == SIGKILL)
```

The **WTERMSIG** macro performs conversion of OpenEdition signal numbers to SAS/C signal numbers.

EXAMPLE

The following example illustrates the use of **waitpid** to wait for a process to end:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int i, status;
   pid_t childID, endID;
   time_t when;

   if ((childID = fork()) == -1) {     /* Start a child process.      */
      perror("fork error");
      exit(EXIT_FAILURE);
   }
   else if (childID == 0) {            /* This is the child.          */
      time(&when);
      printf("Child process started at %s", ctime(&when));
      sleep(10);                       /* Sleep for 10 seconds.       */
      exit(EXIT_SUCCESS);
   }
   else {                              /* This is the parent.         */
      time(&when);
      printf("Parent process started at %s", ctime(&when));

         /* Wait 15 seconds for child process to terminate.          */
      for(i = 0; i < 15; i++) {
         endID = waitpid(childID, &status, WNOHANG|WUNTRACED);
         if (endID == -1) {            /* error calling waitpid       */
            perror("waitpid error");
            exit(EXIT_FAILURE);
         }
         else if (endID == 0) {        /* child still running         */
            time(&when);
            printf("Parent waiting for child at %s", ctime(&when));
            sleep(1);
```

waitpid Wait for a Specific Process to End
(continued)

```
                }
                else if (endID == childID) {  /* child ended                */
                   if (WIFEXITED(status))
                      printf("Child ended normally.\n");
                   else if (WIFSIGNALED(status))
                      printf("Child ended because of an uncaught signal.\n");
                   else if (WIFSTOPPED(status))
                      printf("Child process has stopped.\n");
                   exit(EXIT_SUCCESS);
                }
             }
          }
       }
```

RELATED FUNCTIONS

fork, **wait**

w_getmntent Get Mounted File System Information

SYNOPSIS

```
#include <sys/mntent.h>

int w_getmntent(void *sysInfo, int infoSize);
```

DESCRIPTION

w_getmntent gets information about the mounted file systems and saves that information in **w_mntent** structures, declared in **<sys/mntent.h>**. The arguments to **w_getmntent** are:

sysInfo
is a pointer to a buffer area used to save the file system information. The **w_getmntent** function copies a **w_mntent** structure into this area for each file system that is mounted.

infoSize
specifies the size of the **sysInfo** buffer area. If you are not sure of the required size, you can specify a **0**, which will cause the **w_getmntent** function to return the number of file systems without actually copying any **w_mntent** information.

The **w_mntent** structures contains entries that provide the following information:

mnt_fsname
is the name of a mounted file system. This name can be up to 45 characters in length and ends with a NULL character. You can use this name with the **w_statfs** function to obtain more information.

mnt_dir
is the null-terminated directory name for the mounted file system.

A **w_mnth** structure, which is also declared in **<sys/mntent.h>**, is stored at the beginning to the **sysInfo** buffer area, before any of the **w_mntent** structues. The **w_mnth** structure forms a buffer area header that contains positioning information that is used when multiple calls to **w_getmntent** are made to store information about several mounted file systems. The following **w_mnth** entries are used:

mnth_size
is the size in bytes of the buffer area.

mnth_cur
is the current position in the buffer area.

The positioning information should be initialized to **0**s and should not be changed between calls to the **w_getmntent** function. Subsequent calls to **w_getmntent** will append new **w_mntent** structures to the buffer area. A **0** is returned by **w_getmntent** after information about the last file system has been written to the buffer area.

RETURN VALUE

If successful, **w_getmntent** returns the number of **w_mntent** structures that have been copied to the buffer area. If **infoSize** was **0**, the total number of file

w_getmntent Get Mounted File System Information

(continued)

systems is returned, but no data are copied. A **-1** is returned if **w_getmntent** is unsuccessful.

PORTABILITY

The **w_getmntent** function may be useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

EXAMPLE

The following example illustrates the use of **w_getmntent** to obtain information about a mounted file system:

```
#include <sys/types.h>
#include <sys/mntent.h>
#include <string.h>
#include <stdio.h>

#define MAX_FILE_SYS 10

main()
{
   int lastSys = 0, sysCount;

   struct {
      struct w_mnth header;
      struct w_mntent fileSysInfo[MAX_FILE_SYS];
   } bufferArea;

   memset(&bufferArea, 0, sizeof(bufferArea));

   do {
      lastSys = w_getmntent((char *)
                &bufferArea, sizeof(bufferArea));
      if (lastSys == -1)
         perror("Error during call to w_getmntent.");
      else {
         for (sysCount = 0; sysCount < lastSys; ++ sysCount) {
            printf("File System = %s\n",
                  bufferArea.fileSysInfo[sysCount].mnt_fsname);
            printf("Mount Point = %s\n\n",
                  bufferArea.fileSysInfo[sysCount].mnt_mountpoint);
          }
      }
   } while (lastSys > 0);
}
```

RELATED FUNCTIONS

mount, **w_statfs**

w_getpsent Get Process Information

SYNOPSIS

```
#include <sys/ps.h>

int w_getpsent(int processPos, struct w_psproc *bufferArea,
               size_t bufferLength);
```

DESCRIPTION

w_getpsent gets information about all processes for which the caller is authorized. The arguments to the **w_getpsent** function are:

processPos
is an integer that identifies the relative position of a process within the system, with a **0** representing the first process. You should call **w_getpsent** from within a loop. On the first call, pass a **0** as the **processPos** argument, and **w_getpsent** will return the position value of the next process that the calling process has access to. This value is then used as the **processPos** argument for the next call. Continue to loop until a **0** is returned.

bufferArea
is a pointer to a buffer area that is used to store information about the processes.

bufferLength
is the length in bytes of the buffer area.

The information about a process is stored in a **w_psproc** structure, which is declared in **<sys/ps.h>**. You may access the following elements of this structure to obtain process information:

unsigned int ps_state
process state

pid_t ps_pid
process identification

pid_t ps_ppid
process parent identification

pid_t ps_sid
session identification

pid_t ps_pgpid
process group identification

pid_t ps_fgpid
foreground process group identification

uid_t ps_euid
effective user identification

uid_t ps_ruid
real user identification

uid_t ps_suid
saved user identification

gid_t ps_egid
effective group identification

w_getpsent Get Process Information
(continued)

gid_t ps_rgid
real group identification

gid_t ps_sgid
saved set group identification

long ps_size
total size

time_t ps_starttime
start time

clock_t ps_usertime
user CPU time

clock_t ps_systime
system CPU time

int ps_conttylen
controlling terminal name length

char *ps_conttyptr
controlling terminal name

int ps_pathlen
length of pathname

char *ps_pathptr
pathname

int ps_cmdlen
length of command

char *ps_cmdptr
command and arguments

RETURN VALUE

If successful, **w_getpsent** returns an integer that identifies the next process that the calling process has access to. A **0** is returned to indicate that there are no more accessible processes. A **-1** is returned if the call to **w_getpsent** is unsuccessful.

PORTABILITY

The **w_getpsent** function may be useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

EXAMPLE

The following example illustrates the use of **w_getpsent** to obtain process information:

```
#include <sys/types.h>
#include <sys/ps.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_CHAR 256
```

w_getpsent Get Process Information

(continued)

```
main()
{
   int processNum = 0;
   W_PSPROC workArea;

      /* Initialize work area.                    */
   memset(&workArea, 0, sizeof(workArea));

      /* Allocate memory for character strings.  */
   if ((workArea.ps_conttyptr = (char *) malloc(MAX_CHAR)) == NULL) {
      perror("ps_conttyptr memory allocation error");
      abort();
   }
   else
      workArea.ps_conttylen = MAX_CHAR;

   if ((workArea.ps_pathptr = (char *) malloc(MAX_CHAR)) == NULL) {
      perror("ps_pathptr memory allocation error");
      abort();
   }
   else
      workArea.ps_pathlen = MAX_CHAR;

   if ((workArea.ps_cmdptr = (char *) malloc(MAX_CHAR)) == NULL) {
      perror("ps_cmdptr memory allocation error");
      abort();
   }
   else
      workArea.ps_cmdlen = MAX_CHAR;

      /* Get process information.                 */
   do {
      processNum = w_getpsent(processNum, &workArea, sizeof(workArea));
      if (processNum == -1)
         perror("error in w_getpsent");
      else  {
         printf("process number = %d\n", processNum);
         printf("process ID = %10d\n", workArea.ps_pid);
         printf("User ID = %8u\n\n", workArea.ps_ruid);
      }
   } while (processNum != 0 && processNum != 1);
}
```

w_ioctl Pass Command and Arguments to an I/O Device

SYNOPSIS

```
#include <termios.h>

int w_ioctl(int fileDescriptor, int command, int argLength,
            void *argBuffer);
```

DESCRIPTION

w_ioctl enables you to pass device-specific commands and arguments to an I/O device driver. **w_ioctl** accepts the following arguments:

fileDescriptor
is a file descriptor for a file associated with a device driver.

command
is an integer that specifies a device-specific command that is passed to the device driver.

argLength
specifies the length in bytes of the **arg** argument. The minimum length is 1 and maximum length is 1024.

argBuffer
is a pointer to the buffer that holds the arguments to be passed to the device driver.

RETURN VALUE

w_ioctl returns a **0** if successful and a **-1** if unsuccessful.

PORTABILITY

The **w_ioctl** function may be useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

EXAMPLE

The following example illustrates the use of **w_ioctl** to pass a command to the controlling terminal:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define MAX_BUFFER 1024
#define BREAK 0x00
#define DEV_SPECIFIC_CMD 1

main()
{
   char parameters[MAX_BUFFER];
   int result;
```

w_ioctl Pass Command and Arguments to an I/O Device
(continued)

```
        /* Initialize parameter buffer.                        */
    memset(parameters, BREAK, sizeof(parameters));

        /* Pass command and parameter to stdin device driver. */
    if ((result = w_ioctl(STDIN_FILENO, DEV_SPECIFIC_CMD,
         sizeof(parameters), parameters)) != 0)
        perror("I/O control error");
    else
        printf("SUCCESSn");
}
```

RELATED FUNCTIONS

tcdrain, **tcflow**, **tcflush**, **tcsendbreak**

w_statfs Get File System Status Information

SYNOPSIS

```
#include <sys/statfs.h>

int w_statfs(const char *fileSystem, struct w_statfs *statInfo,
             size_t infoSize);
```

DESCRIPTION

w_statfs gets file system status information. The following arguments are required:

fileSystem
is the name of the file system for which status information is to be provided. File system names can be obtained by using the **w_getmntent** function.

statInfo
is a pointer to a **w_statfs** structure that is declared in the **<sys/statfs.h>** header. The following elements of this structure provide file system status information:

int statfs_len
is the size of the **w_statfs** structure.

int statfs_blksize
is the block size for the file system.

unsigned int statfs_total_space
is the total number of blocks used by the file system.

unsigned int statfs_free_space
is the number of free blocks available to unprivileged users.

infoSize
specifies the number of bytes of information to be stored. If **infoSize** is larger than the size of the **w_statfs** structure, then all of the information is stored; if it is shorter than the structure, then only a portion of the information is stored; and if **infoSize** is **0**, then no information is stored. A value of **0** for **infoSize** may be used to test whether or not a file system exists.

RETURN VALUE

If successful, **w_statfs** returns the number of bytes stored. If **infoSize** is **0**, the total number of bytes of information available is returned. A **-1** is returned if unsuccessful.

PORTABILITY

The **w_statfs** function may be useful in OpenEdition applications; however, it is not defined by the POSIX.1 standard and should not be used in portable applications.

w_statfs Get File System Status Information
(continued)

EXAMPLE

The following example illustrates the use of **w_statfs** to obtain file system status information:

```
#include <sys/types.h>
#include <sys/statfs.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>

main()
{
   const char fileSys[] = "POSIX.FILE.SYSTEM";
   char *mountPoint = "/u/userid/dev";
   char *mountType = "HFS     ";
   struct w_statfs workArea;

      /* Initialize work area.               */
   memset(&workArea, 0x00, sizeof(workArea));

      /* Mount file system.                  */
   if (mount(mountPoint, fileSys, mountType, MTM_RDWR, 0, NULL) != 0)
      perror("error mounting file system");

      /* Get file system status information. */
   if (w_statfs(fileSys, &workArea, sizeof(workArea)) == -1)
      perror("w_statfs error");
   else
      printf("Length = %d\n", workArea.statfs_len);
      printf("Block Size = %d\n", workArea.statfs_blksize);
      printf("Blocks Used = %u\n", workArea.statfs_total_space);
      printf("Blocks Allocated = %u\n", workArea.statfs_used_space);
      printf("Free Space = %u\n", workArea.statfs_free_space);
}
```

RELATED FUNCTIONS

w_statfs, **w_getmntent**

Function Index

ABEND . . . . . Abnormally Terminate a Program 4-4

ATTACH . . . . . Create a New Subtask 4-5

CHAP . . . . . Change a Task's Priority 4-8

CMSSTOR . . . . . Allocate or Free Storage 4-9

DEQ . . . . . Release an Enqueued Resource 4-14

DETACH . . . . . Remove a Subtask from the System 4-13

DMSFREE . . . . . Allocate or Free DMSFREE Storage 4-11

ENQ . . . . . Wait for a Resource to Become Available 4-15

ESTAE . . . . . Define an Abnormal Termination Exit 4-16

GETLONG, _getlong . . . . . Gets a Long Integer from a Character Buffer 18-32

GETMAIN/FREEMAIN . . . Allocate or Free Virtual Storage 4-18

GETSHORT, _getshort . . . . . Gets a Short Integer from Character Buffer 18-49

IUCV . . . . . Perform IUCV Functions 5-12

POST . . . . . Post an ECB 4-20

PUTLONG, putlong . . . . . Puts a Long Integer into a Character Buffer 18-74

PUTSHORT, putshort . . . . . Puts a Short Integer into a Character Buffer 18-75

RDTERM . . . . . Simulate the RDTERM Assembler Language Macro 4-21

SETRP . . . . . Set Recovery Parameters in an ESTAE Exit 4-22

STATUS . . . . . Halt or Resume a Subtask 4-23

STIMER . . . . . Define a Timer Interval 4-24

STIMERM**...** . . . . . Define, Test, or Cancel a Real Time Interval 4-25

TPUT/TGET . . . . . Terminal I/O via SVC 93 4-27

TTIMER . . . . . Test or Cancel a Timer Interval 4-30

WAIT . . . . . Wait for the POST of One or More ECB's 4-33

WAITT . . . . . Simulate the WAITT Assembler Language Macro 4-37

WRTERM . . . . . Simulate the WRTERM Assembly Language Macro 4-38

__passwd . . . . . Verify or Change a Password 20-66

_exit . . . . . End Process and Skip Cleanup 20-29

accept . . . . . Accepts a Connection Request 18-2

addsrch . . . . . Indicate a ''Location'' from which Modules May Be Loaded 1-3

appc . . . . . Perform APPC/VM Functions 6-6

appcconn . . . . . Perform an APPC/VM CONNECT 6-5

appcsevr . . . . . Perform an APPC/VM SEVER 6-7

atcoexit . . . . . Register Coprocess Cleanup Function 9-12

atfork . . . . . Define Fork Exits 20-2
bind . . . . . Assigns a Name to a Socket 18-6
buildm . . . . . Create a Function Pointer from an Address 1-5
cfgetispeed . . . . . Determine Input Baud Rate 20-5
cfgetospeed . . . . . Determine Output Baud Rate 20-7
cfsetispeed . . . . . Set Input Baud Rate 20-9
cfsetospeed . . . . . Set Output Baud Rate 20-11
chaudit . . . . . Change File Audit Flags (Using Pathname) 20-13
chown . . . . . Change File Owner or Group (Using Pathname) 20-15
cmspid . . . . . Tokenize a CMS Fileid 2-9
cmsrxfn . . . . . Create a REXX Function Package 8-7
cmsshv . . . . . Fetch, Set, or Drop REXX or EXEC2 Variable Values 8-10
cmsstack . . . . . Insert a String into the CMS Program Stack 8-14
cocall . . . . . Pass Control to a Coprocess 9-14
coexit . . . . . Terminate Coprocess Execution 9-16
connect . . . . . Associates a Socket with a Process 18-8
coproc . . . . . Return the ID of a Coprocess 9-17
coreturn . . . . . Return Control to a Coprocess 9-18
cosignal . . . . . Define a Global Signal Handler 9-20
costart . . . . . Create a New Coprocess 9-23
costat . . . . . Return Coprocess Status 9-25
delsrch . . . . . Delete a ‘‘Location’’ in the Load Module Search Order List 1-7
dn_comp . . . . . Translates Domain Names to Compressed Format 18-10
dn_expand . . . . . Expands Compressed Domain Names 18-11
endhostent . . . . . Closes a Host File or TCP Connection 18-12
endnetent . . . . . Closes the Network File 18-13
endprotoent . . . . . Closes the Protocol File 18-14
endrpcent . . . . . Closes the `/etc/rpc` Protocol File 18-15
endservent . . . . . Closes the Services File 18-16
envlevel . . . . . Get Subenvironment Information 13-4
envname . . . . . Get Subenvironment Name 13-5
execcall . . . . . Identify a Macro to Be Executed 7-6
execend . . . . . Cancel the SUBCOM Interface 7-8
execget . . . . . Return the Next Subcommand 7-10
execid . . . . . Parse a Line of Input as a Subcommand 7-13
execinit . . . . . Create a SUBCOM Environment 7-15
execl . . . . . Overlay Calling Process and Run New Program 20-17

execle . . . . . Overlay Calling Process and Run New Program 20-19
execlp . . . . . Overlay Calling Process and Run New Program 20-21
execmsg . . . . . Send a Message to the Terminal 7-17
execmsi . . . . . Return System Message Preference 7-19
execrc . . . . . Set Return Code of Most Recent Subcommand 7-21
execshv . . . . . Fetch, Set, or Drop REXX, EXEC2, or CLIST Variable Values 7-22
execv . . . . . Overlay Calling Process and Run New Program 20-23
execve . . . . . Overlay Calling Process and Run New Program 20-25
execvp . . . . . Overlay Calling Process and Run New Program 20-27
fchaudit . . . . . Change File Audit Flags (Using File Descriptor) 20-30
fchown . . . . . Change File Owner or Group (Using File Descriptor) 20-32
fcntl . . . . . Controls Socket Operating Characteristics 18-17
fork . . . . . Create a New Process 20-34
fpathconf . . . . . Determine Pathname Variables 20-37
getclientid . . . . . Gets the Calling Application Identifier 18-18
getdtablesize . . . . . Gets Descriptor Table Size 18-20
getegid . . . . . Determine Effective Group ID 20-39
geteuid . . . . . Determine Effective User ID 20-40
getgid . . . . . Determine Real Group ID 20-41
getgrgid . . . . . Access Group Database by ID 20-42
getgrnam . . . . . Access Group Database by Name 20-44
getgroups . . . . . Determine Supplementary Group IDs 20-46
getgroupsbyname . . . . . Determine Supplementary Group IDs for a User Name 20-48
gethostbyaddr . . . . . Gets Host Information by Address 18-22
gethostbyname . . . . . Gets Host Information by Name 18-23
gethostent . . . . . Gets the Next Entry in the Host File 18-25
gethostid . . . . . Gets the Local Host's Internet Address 18-28
gethostname . . . . . Gets the Host Name 18-30
getnetbyaddr . . . . . Gets Network Information by Address 18-33
getnetbyname . . . . . Gets Network Information by Name 18-34
getnetent . . . . . Gets the Next Network Information Structure 18-35
getpeername . . . . . Gets the Address of a Peer 18-36
getpgrp . . . . . Determine Process Group ID 20-50
getpid . . . . . Determine Process ID 20-51
getppid . . . . . Determine Parent Process ID 20-52
getprotobyname . . . . . Gets Protocol Information by Name 18-38
getprotobynumber . . . . . Gets Protocol Information by Number 18-39

getprotoent . . . . . Gets Protocol Information 18-40
getpwnam . . . . . Access User Database by User Name 20-53
getpwuid . . . . . Access User Database by User ID 20-55
getrpcbyname . . . . . Returns **rpcent** Structure for an RPC Program Name 18-41
getrpcbynumber . . . . . Returns the **rpcent** Structure for an RPC Program Number 18-42
getrpcent . . . . . Returns the **rpcent** Structure 18-43
getservbyname . . . . . Gets Service Information by Name 18-44
getservbyport . . . . . Gets Service Information by Port 18-46
getservent . . . . . Gets Next Entry in Services File 18-47
getsockname . . . . . Gets a Socket by Name 18-50
getsockopt . . . . . Gets the Value of an Option 18-52
getuid . . . . . Determine Real User ID 20-57
givesocket . . . . . Gives a Socket to Another Process 18-55
herror . . . . . Prints a Host Error Message 18-57
htoncs . . . . . Converts an EBCDIC Character to ASCII 18-58
htonl . . . . . Converts a Long Integer from Host to Network Order 18-59
htons . . . . . Converts a Short Integer from Host to Network Order 18-60
inet_addr . . . . . Interprets an Internet Address 18-61
inet_lnaof . . . . . Determines a Local Network Address 18-63
inet_makeaddr . . . . . Constructs an Internet Address 18-64
inet_netof . . . . . Determines the Network Number 18-65
inet_network . . . . . Interprets an Internet Network Number 18-66
inet_ntoa . . . . . Puts an Internet Address into Network Byte Order 18-67
initgroups . . . . . Initialize Supplementary Group IDs for a Process 20-58
intractv . . . . . Indicate Interactive Execution 13-6
ioctl . . . . . Controls Operating Characteristics of Socket Descriptors 18-68
iucvacc . . . . . Perform an IUCV ACCEPT 5-9
iucvclr . . . . . Terminate IUCV Communications 5-10
iucvconn . . . . . Perform an IUCV CONNECT 5-11
iucvset . . . . . Initialize IUCV Communications 5-14
iucvsevr . . . . . Perform an IUCV SEVER 5-15
listen . . . . . Indicates Socket Descriptor is Ready to Accept Requests 18-70
loadd . . . . . Dynamically Load a Load Module Containing Data 1-8
loadm . . . . . Dynamically Load a Load Module 1-10
localeconv . . . . . Get Numeric Formatting Convention Information 10-6
mblen . . . . . Determine Length of a Multibyte Character 11-5
mbstowcs . . . . . Convert a Multibyte Character Sequence to a Wide Sequence 11-7

mbtowc . . . . . Convert a Multibyte Character to a Wide Character 11-9
mknod . . . . . Create a Character or FIFO Special File 20-59
mount . . . . . Mount a File System 20-61
ntohcs . . . . . Converts ASCII Characters to EBCDIC 18-71
ntohl . . . . . Converts a Long Integer from Network to Host Byte Order 18-72
ntohs . . . . . Converts a Short Integer from Network to Host Byte Order 18-73
oeattach . . . . . Create a Child Process as a Subtask 20-63
oeattache . . . . . Create a Child Process as a Subtask 20-65
pathconf . . . . . Determine Pathname Variables that Can Be Configured 20-67
readv . . . . . Reads Data from a Socket Descriptor into an Array of Buffers 18-76
recv . . . . . Stores Messages from a Connected Socket 18-77
recvfrom . . . . . Stores Messages from a Connected or Unconnected Socket Descriptor 18-78
recvmsg . . . . . Receives Data from a Connected or Unconnected Socket Descriptor 18-80
res_init . . . . . Initializes the Resolver 18-82
res_mkquery . . . . . Makes a Domain Name Server Packet 18-84
res_send . . . . . Sends a Query 18-86
rxeval . . . . . Return a Result Value to REXX 8-15
rxresult . . . . . Return a Result Value to REXX 8-16
select . . . . . Determines the Number of Ready Sockets 18-87
selectecb . . . . . Determines the Number of Ready Sockets 18-90
send . . . . . Sends a Message to a Connected Socket 18-92
sendmsg . . . . . Sends a Message to a Connected or Unconnected Socket 18-93
sendto . . . . . Sends a Message to a Socket 18-96
setegid . . . . . Specify Effective Group ID 20-69
seteuid . . . . . Specify Effective User ID 20-70
setgid . . . . . Specify Group ID 20-71
setgroups . . . . . Set Supplementary Group IDs 20-72
sethostent . . . . . Opens a Host File or Connects to a Name Server 18-97
setlocale . . . . . Select Current Locale 10-9
setnetent . . . . . Opens the Network File 18-98
setpgid . . . . . Specify Process Group ID for Job Control 20-73
setprotoent . . . . . Opens the Protocols File 18-99
setrpcent . . . . . Opens the `/etc/rpc` Protocols File 18-100
setservent . . . . . Opens the Services File 18-101
setsid . . . . . Create Session and Specify Process Group ID 20-77
setsockimp . . . . . Specifies the Socket Implementation 18-102
setsockopt . . . . . Sets Socket Options 18-103

setuid . . . . . Specify User ID 20-78
shutdown . . . . . Ends Communication with a Socket 18-105
sigdef . . . . . Define User Signal 12-5
socket . . . . . Creates a Socket 18-106
socketpair . . . . . Creates a Connected Socket Pair 18-111
socktrm . . . . . Terminates the Socket Library Environment 18-112
strcoll . . . . . Compare Two Character Strings Using Collating Sequence 10-13
strxfrm . . . . . Transform a String Using Locale-Dependent Information 10-15
sysconf . . . . . Determine System Configuration Options 20-79
syslevel . . . . . Get Operating System Information 13-8
sysname . . . . . Get Operating System Name 13-9
takesocket . . . . . Takes a Socket Descriptor from Donor Process 18-113
tcdrain . . . . . Wait for Output to Drain 20-81
tcflow . . . . . Controls the Flow of Data to a Terminal 20-83
tcflush . . . . . Flush Terminal Input or Output 20-85
tcgetattr . . . . . Get Terminal Attributes 20-87
tcgetpgrp . . . . . Get Foreground Process Group Identification 20-89
tcsendbreak . . . . . Send a Break Condition 20-91
tcsetattr . . . . . Set Terminal Attributes 20-93
tcsetpgrp . . . . . Set Foreground Process Group Identification 20-96
times . . . . . Determine Process Times 20-98
typlin . . . . . Invoke the CMS TYPLIN Function 4-31
umask . . . . . Change File Mode Creation Mask 20-101
umount . . . . . Unmounts a File System 20-103
uname . . . . . Display Current Operating System Name 20-105
unloadd . . . . . Discard a Previously Loaded Data Module 1-17
unloadm . . . . . Discard a Previously Loaded Module 1-18
w_getmntent . . . . . Get Mounted File System Information 20-114
w_getpsent . . . . . Get Process Information 20-116
w_ioctl . . . . . Pass Command and Arguments to an I/O Device 20-119
w_statfs . . . . . Get File System Status Information 20-121
wait . . . . . Wait for Child Process to End 20-107
waitpid . . . . . Wait for a Specific Process to End 20-110
waitrd . . . . . Invoke the CMS WAITRD Function 4-34
wcstombs . . . . . Convert a Wide Character Sequence to a Multibyte Sequence 11-11
wctomb . . . . . Convert a Wide Character to a Multibyte Character 11-12
writev . . . . . Writes Data from an Array of Buffers to a Socket 18-114