IBM RT PC Advanced Interactive Operating System Version 2.1

AIX Operating System Technical Reference

Volume 1

Programming Family

Personal
Computer
Software

SC23-0808-0

IBM RT PC Advanced Interactive Operating System Version 2.1

AIX Operating System Technical Reference

Volume 1

Programming Family

IBM

Personal
Computer
Software

SC23-0808-0

First Edition (January 1987)

Portions of the code and documentation described in this book were developed at the Electrical Engineering and Computer Sciences Department at the Berkeley Campus of the University of California under the auspices of the Regents of the University of California.

This edition applies to Version 2.1 of the Advanced Interactive Executive Operating System, and to all subsequent releases until otherwise indicated in new editions or technical newsletters. Changes are made periodically to the information herein; these changes will be reported in technical newsletters or in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is," without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address given below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer or your IBM marketing representative.

A reader's comment form is provided at the back of this publication. If the form has been removed, address comments to IBM Corporation, Department 997, 11400 Burnet Road, Austin, Texas 78758-3493. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

About This Book

This book provides information about the programming interface to the Advanced Interactive Executive Operating System (AIX).[1] This information is needed to write applications and systems software for AIX.

Who Should Use This Book

This book is intended for experienced C programmers. To use this book effectively, you should be familiar with AIX or UNIX[2] System V commands, system calls, subroutines, file formats, and special files. If you are not already familiar with AIX or UNIX System V, see *Using the AIX Operating System.*

How To Use This Book

Chapters 2, 3, 4, 5, 6, and 7 are organized in alphabetical order by name. Related information in those chapters is combined into one description where applicable, but each item appears as a separate entry in the Index.

Organization

This book is divided into the two volumes, Volume 1 and Volume 2. Each volume contains a complete table of contents, list of figures, and index for both volumes.

Volume 1 contains Chapters 1 through 3; Volume 2 contains Chapters 4 through 7 and Appendixes A through F. The subjects of these chapters are:

- Chapter 1, "AIX Operating System," gives an overview of the various subsystems discussed in this book.
- Chapter 2, "System Calls," describes the C Language interface to the operating system calls, which are also called supervisor calls (SVCs) and kernel primitives.

[1] Advanced Interactive Executive and AIX are trademarks of International Business Machines Corporation.

[2] UNIX was developed and licensed by AT&T. It is a registered trademark of AT&T in the United States of America and other countries.

- Chapter 3, "Subroutines," lists subroutines and macros available to C Language programmers.
- Chapter 4, "File Formats," defines the formats of various system and user files.
- Chapter 5, "Miscellaneous Facilities," includes miscellaneous information, such as text-processing macro packages.
- Chapter 6, "Special Files," describes special files associated with specific peripheral devices and device drivers.
- Chapter 7, "Advanced Display Graphics Support Library," describes the Advanced Display Graphics Support Library.

In the chapters that are organized in alphabetical order, some entries describe several facilities (system calls, subroutines, file formats, or special files). In such cases, each facility appears only once, alphabetized under its *major* name. If you have difficulty finding a given facility, look it up in the index at the back of the book.

All entries are based on a common format, but all of the sections do not always appear.

Purpose
Briefly describes the purpose of the facility.

Syntax or **Synopsis**
Shows how to use the facility. See "Syntax" on page v for details about the information given in the "Syntax" sections for system calls and subroutines.

Description
Describes the system call, subroutine, file format, or special file in detail.

Return Value
Explains the value returned by a system call or subroutine.

Diagnostics
Lists the possible values that a system call can return in the **errno** external variable if an error occurs. See Appendix A, "Error Codes," for a complete list of these values.

Examples
Shows one or more examples of how to use the facility.

Files
Lists the names of files that the facility uses.

Related Information
Refers you to additional information you may find helpful. This section refers you first to additional topics in this book, then to other publications. References from this book are given in the order they appear in the book, alphabetically within each chapter.

Volume 2 also contains the following appendixes:

- Appendix A, "Error Codes," lists and describes the values that system calls pass back when they encounter error conditions.
- Appendix B, "Writing a Queuing System Backend," provides detailed information about writing friendly backend programs.
- Appendix C, "Writing Device Drivers," gives information needed to write and install AIX device drivers.
- Appendix D, "Porting DOS 3.0 Applications," explains how to port programs from the IBM Personal Computer Disk Operating System (DOS) to the AIX Operating System.
- Appendix E, "Component Cross Reference," lists the programs with which certain subroutines and subroutine libraries are packaged.
- Appendix F, "Glossary," defines terms to help you better understand AIX and the RT PC.

A Reader's Comment Form and Book Evaluation Form are provided at the back of each volume of this book. Use the Reader's Comment Form at any time to give IBM information that may improve the book. After you have become familiar with the book, use the Book Evaluation Form to give IBM specific feedback about the book.

Syntax

The "Syntax" section of each system call and subroutine entry in this book gives the syntax needed to invoke it. The following conventions are used in this section:

Boldface type shows text to be entered exactly as shown.

Italic type shows parameters that should be replaced with actual values.

[ ] (square brackets) enclose optional parameters.

. . . (an ellipsis) follows a parameter that can be repeated any number of times.

The information shown in each "Syntax" section is usually the set of declarations as they might appear in the actual C-language definition of the call or subroutine. These declarations give you more information than showing the exact calling sequence as it appears in a user program.

Consider the following example of a subroutine entry:

```
#include <stdio.h>

FILE *fopen (path, type)
char *path, *type;
```

The **#include** statement names a header file that contains definitions needed by the subroutine. See "Header Files" on page vii for more information.

The first line following the **#include** statement shows the data type of the return value (**FILE ***), the name of the subroutine (**fopen**), and the parameters that it takes (*path* and *type*). The following lines indicate the data type of each parameter. The fact that the name **FILE** is in all capitals indicates that this data type is defined in the **stdio.h** header file.

This subroutine might actually be used in a program like this:

```
#include <stdio.h>
 . . .
main ( )
{
   FILE *inputfile;
   char filename[ ] = "test.data";
    . . .
   inputfile = fopen (filename, "r+");
    . . .
}
```

Note that the type of both parameters is stated as **char *** (pointer to character), but that the value given for each is actually a pointer to a character string (an array of characters). In the C language, pointers and arrays are treated similarly so that the notations `*p` and `p[0]` are generally interchangeable. Thus, when this book shows a parameter of type **char ***, a character string is frequently required. Check the "Description" section to make sure.

Because **fopen** returns a type other than **int**, the subroutine must be declared so that the compiler knows this information. In this particular case, it is already declared for you in the header file. Sometimes, however, you may need to declare a system call or subroutine yourself. For this example, the declaration would take the form:

```
FILE *fopen();
```

Such a declaration should be put at the top of your program, before any references to the system call or subroutine. Note that the declaration resembles the syntax shown in this book, except that the parameters are omitted and a semicolon is added to the end.

See *C Language Guide and Reference* or another C language manual for more detailed information about pointers, arrays, and subroutine declarations.

Header Files

Many system calls and subroutines require that header files be included in the programs that use them. When this is the case, the **#include** statements needed are shown in the "Syntax" section of the call or subroutine entry. Consider the following example:

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

When a program is being compiled, this **#include** statement inserts the text of the **stdio.h** header file into the source program. The < > delimiters indicate that the file is located in the **/usr/include** directory. All of the header files used by the system calls and subroutines described in this book are located in **/usr/include** or in one of its subdirectories.

The header files contain definitions of constants and macros that the C language preprocessor interprets. The **#include** statements must precede all references in your program to the constants and macros that the header files define. Most of the time you can simply put all of the **#include** statements together at the top of the program. If you use several calls or subroutines that require the same header files, then each file should be included only once. If a system call or subroutine requires more than one header file, be careful to enter the **#include** statements in the order shown.

By convention, the names of most of the constants are in capital letters. Therefore, a name that appears in this book in bold capitals (for example, **EFAULT**) is a constant defined in a header file. A few constants are not named in capitals, notably **stdin**, **stdout**, and **stderr**, which are defined in **stdio.h**.

The constant **NULL** is commonly used to denote a null pointer value. This book sometimes mentions **NULL** when discussing system calls and subroutines that do not require header files. If you compile a program and get an error message indicating that **NULL** is not defined, insert the following statement before the first **NULL** in your program:

```
#define  NULL  0
```

In addition to constants, header files sometimes define macros and data types. Macros take parameters and resemble subroutines, but there are several differences:

- You must not type a space between the macro name and the open parenthesis that follows it. For example, the macro call `getc( . . . )` is valid, but `getc ( . . . )` is not. The preprocessor does not recognize the second of these as a macro, and so it does not make the proper substitution.
- You cannot take the address of a macro with the C language & operator.
- The parameters of a macro are evaluated in a different manner from those of a subroutine. See *C Language Guide and Reference* or another C language manual for details about parameter evaluation.

This book describes a macro as a subroutine, then adds statement that it is implemented as a macro.

Related Publications

Related information can be found in the following books:

- *IBM RT PC Using the AIX Operating System* describes using the AIX Operating System commands, working with file systems, and developing shell procedures.
- *IBM RT PC Managing the AIX Operating System* provides instructions for performing such system management tasks as adding and deleting user IDs, creating and mounting file systems, and repairing file system damage.
- *IBM RT PC AIX Operating System Programming Tools and Interfaces* describes the programming environment of the AIX Operating System and includes information about using the operating system tools to develop, compile, and debug programs. In addition, this book describes the operating system services and how to take advantage of them in a program. This book also includes a diskette that includes programming examples, written in C language, to illustrate using system calls and subroutines in short, working programs. (Available optionally)
- *IBM RT PC AIX Operating System Commands Reference* lists and describes the AIX Operating System commands.
- *IBM RT PC C Language Guide and Reference* provides guide information for writing, compiling, and running C language programs and includes reference information about C language data structures, operators, expressions, and statements. (Available optionally)
- *IBM RT PC Assembler Language Reference* describes the IBM RT PC Assembler Language and the 032 Microprocessor and includes descriptions of syntax and semantics, machine instructions, and pseudo-operations. This book also shows how to link and run Assembler Language programs, including linking to programs written in C language. (Available optionally)
- *IBM RT PC Keyboard Description and Character Reference* describes the national character and keyboard support for the 101-key, 102-key, and 106-key keyboards, including keyboard position codes, keyboard states, control code points, code sequence processing, and nonspacing character sequences.
- *IBM RT PC Virtual Resource Manager Technical Reference* is a two-volume set. The first volume, *Virtual Resource Manager Programming Reference*, describes the VRM programming environment, including the internal VRM routines, VRM floating-point support, use of the VRM debugger, and the supervisor call instructions that form the Virtual Machine Interface. The second volume, *Virtual Resource Manager Device Support*, describes device IPL and configuration, minidisk management, the virtual terminal and block I/O subsystems, as well as the interfaces to the predefined VRM

device drivers. This volume also describes the programming conventions for developing your own VRM code and installing it on the system.

- *IBM RT PC Hardware Technical Reference* is a three-volume set. Volume I describes how the system unit operates, including I/O interfaces, serial ports, memory interfaces, and CPU interface instructions. Volumes II and III describe adapter interfaces for optional devices and communications and include information about IBM Personal Computer family options and the adapters supported by 6151 and 6150. (Available optionally)
- *IBM RT PC Messages Reference* lists messages displayed by the IBM RT PC and explains how to respond to the messages.
- *IBM RT PC Installing and Customizing the AIX Operating System* provides step-by-step instructions for installing and customizing the AIX Operating System, including how to add or delete devices from the system and how to define device characteristics. This book also explains how to create, delete, or change AIX and non-AIX minidisks.
- *IBM RT PC Personal Computer AT Coprocessor Services Technical Reference* describes differences between writing applications for the IBM Personal Computer AT and IBM RT PC Personal Computer AT Coprocessor Services. This book also includes Personal Computer AT Coprocessor Services BIOS listing, hardware technical information, Math Co-Processor information, and the 286 and 287 instruction sets. (Available optionally)
- *IBM RT PC Using AIX Operating System DOS Services* provides step-by-step information for using AIX Operating System DOS Services. (Available optionally; packaged with *IBM RT PC AIX Operating System DOS Services Reference*)
- *IBM RT PC AIX Operating System DOS Services Reference* provides reference information about the AIX Operating System DOS Services. This book also includes information on sharing DOS files with Personal Computer AT Coprocessor Services, and on the differences between PC DOS and DOS Services. (Available optionally; packaged with *IBM RT PC Using AIX Operating System DOS Services*)

See *IBM RT PC Bibliography and Master Index* for order numbers of RT PC publications and diskettes.

Ordering Additional Copies of This Book

To order additional copies of this publication, use either of the following sources:

- To order from your IBM representative, use Order Number SBOF-0135.
- To order from your IBM dealer, use Part Number 79X3860.

Two binders and the *AIX Operating System Technical Reference* manuals are included with the order. For information on ordering the binders or manuals separately, contact your IBM representative or your IBM dealer.

Contents

Volume 1. System Calls and Subroutines

Chapter 1. AIX Operating System 1-1

Chapter 2. System Calls 2-1

Chapter 3. Subroutines 3-1

Index X-1

Volume 2. Files and Device Drivers

Chapter 4. File Formats 4-1

Chapter 5. Miscellaneous Facilities 5-1

Chapter 6. Special Files 6-1

Chapter 7. Advanced Display Graphics Support Library 7-1

Appendix A. Error Codes A-1

Appendix B. Writing a Queuing System Backend B-1

Appendix C. Writing Device Drivers C-1

Appendix D. Porting DOS 3.0 Applications D-1

Appendix E. Component Cross Reference E-1

Appendix F. Glossary F-1

Index X-1

Figures

3-1. Floating-Point Trap Handler Structures 3-189
3-2. The fpfp Register Mapping 3-190
3-3. Default Error-Handling Procedures 3-240
4-1. Example of Font Storage 4-72
4-2. Information Record Format 4-90
5-1. Octal ASCII Character Set 5-3
5-2. Hexadecimal ASCII Character Set 5-4
5-3. Code Page P0 5-6
5-4. Code Page P1 5-7
5-5. Code Page P2 5-8
5-6. Code Page P0 5-25
5-7. Code Page P1 5-33
5-8. Code Page P2 5-40
5-9. EBCDIC Character Set 5-45
5-10. The eqnchar Characters 5-54
5-11. Greek Characters 5-58
6-1. Bit Positions of ASCII Controls in Echo Map 6-35
6-2. Screen Manager Ring Examples 6-51
6-3. Position Codes for Remapping a Keyboard 6-78
7-1. Default Attribute Values 7-9
E-1. Extended Services Subroutines E-1
E-2. Multi-User Services Subroutines E-1

Volume 1. System Calls and Subroutines

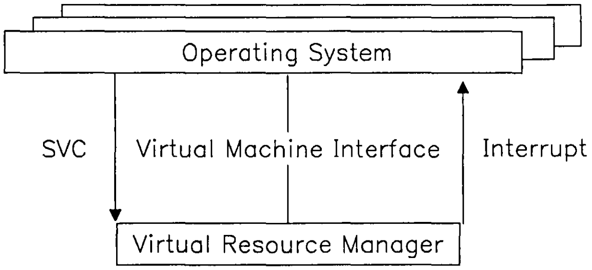
Chapter 1. AIX Operating System

About This Chapter

This overview of the AIX Operating System is divided into three sections. The first section describes process management, creation, and scheduling. The second section is a description of the file system. The final section introduces I/O control and the I/O subsystem.

Overview

The RT PC[1] system software is structured in layers: the Virtual Resource Manager (VRM), the Virtual Machine Interface (VMI), and the AIX Operating System. The VRM extends the hardware function of the processor and memory management to provide a high level of support to hardware devices for the operating system in a virtual machine environment. The VMI is the protocol boundary between the operating system and the VRM. A virtual machine, as defined by the VRM, has a high-level interface that resembles a physical machine. The following figure depicts the software structure and relationships.



Virtual Machine Characteristics

The VRM provides a virtual machine environment that has essentially the same characteristics as the physical machine. Virtual machines run in problem (unprivileged) state and do not directly reflect the supervisory (privileged) characteristics of the physical machine. These functions are handled by the VRM.

A virtual machine has two protection states, user (unprivileged) and operating system (privileged). The operating system state and AIX kernel state are synonymous. Whenever the virtual machine is executing instructions (in either user or operating system state), the processor is actually in problem state. Only the VRM (including code installed in the VRM by the AIX operating system) can execute in real supervisory state.

[1] RT, RT PC, and RT Personal Computer are trademarks of International Business Machines Corporation.

User State

In user state, a virtual machine can issue any of the problem state instructions. One of the instructions available to the virtual machine is the ***supervisor call (SVC)*** instruction, which includes a 16-bit field to indicate which supervisor function is desired. If the high-order bit of this field is set to 1, then the call is made to the VRM; if it is set to 0, then the call is made to the virtual machine supervisor, the AIX Operating System. The VRM cannot be called directly from user state, but an application running in user state can issue an SVC to the virtual machine supervisor and change to operating system state.

Operating System State

A virtual machine enters operating system state either because of an SVC directed to it by a user state process or because of a virtual interrupt directed to it by the VRM. In operating system state, the virtual machine can use all of the instructions available in user state, and can also issue SVCs to the VRM.

Registers

The virtual machine executes in only problem state, so when in user or operating system state it can only directly use the problem state set of registers. The problem state register set consists of the General Purpose Registers (GPRs), the Multiplier Quotient register, the Condition Status register, and five System Control Registers (SCRs), and, if a floating-point processor is present, the floating-point registers. The VRM does permit virtual machines in operating system state to indirectly manipulate the segment registers.

Predefined Memory Locations

Since each virtual machine has a distinct virtual memory space starting at location 0, these locations are valid for all virtual machines. Virtual machine memory locations between 0xC0 and 0x2DC are reserved for communication between the virtual machine and the VRM. These locations are used for memory-mapped timer values, Program Status Blocks (PSBs) for SVCs and interrupts, and miscellaneous other values. Refer to the following table for a summary of predefined virtual memory locations and the values associated with each location.

Memory location Decimal	 Hex	**Bytes** Hex	**Value**
0	0	B8	Reserved
184	B8	4	Number of free paging disk slots
188	BC	4	Number of page replacement cycles
192	C0	4	Number of page faults with preemption
196	C4	4	Number of non-paging disk I/Os
200	C8	2	Reserved
202	CA	2	PCB Segment ID
204	CC	2	Reserved
206	CE	2	Trace buffer synch flag
208	D0	2	VRM sequence number
210	D2	2	Virtual machine sequence number
212	D4	2	Interrupt request buffer
214	D6	1	Process priority
215	D7	1	Floating point register set
216	D8	4	Bus I/O base address
220	DC	4	Bus memory base address
224	E0	4	Virtual interrupt control status
228	E4	2	Execution level
230	E6	2	Virtual machine ID (right justified)
232	E8	4	Time of day, extended
236	EC	4	Virtual timer status
240	F0	4	Real time of day
244	F4	4	Real time of IPL
248	F8	4	Virtual timer source
252	FC	4	Virtual time since IPL

There is a separate PSB for each priority interrupt level, program checks, machine communications and SVC. The PSB includes the Instruction Address Register (IAR) for the point of interrupt, interrupt control and status fields, definition of sublevels and four words of status and data specific to the interrupt.

See *Virtual Resource Manager Technical Reference* for the details of the virtual machine configuration.

VRM Structure

The VRM provides paging support of the virtual memory for the operating system running in a virtual machine. AIX is designed to take advantage of the virtual memory support provided by the VRM. This makes the hardware memory management functions available to AIX while relieving it of the details of paging mechanics, such as page replacement algorithms, management of paging I/O, and so forth.

The interface to virtual memory consists of a set of supervisory call instructions, program check interrupts (addressing and protection exceptions), and machine communication interrupts (page fault occurrence and clearing). The basic model of memory presented to the virtual machine is in terms of segments.

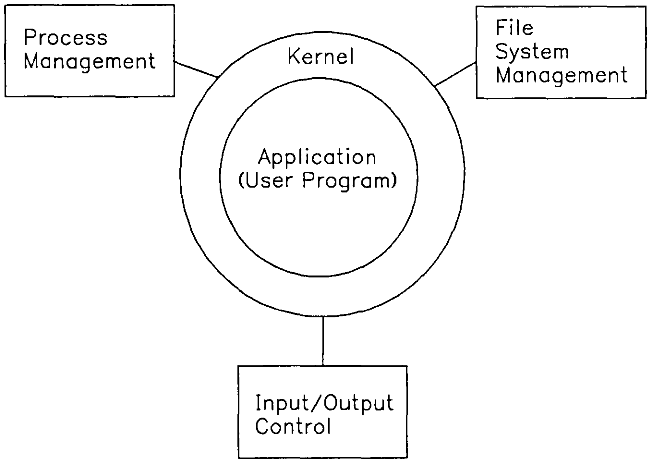
SVCs are provided to create, copy, or destroy segments, and to manipulate segment characteristics. Examples of segment manipulation are changing sizes and protection status, loading segment IDs into the hardware memory management segment registers, and so on. An SVC interface is also provided to allow AIX the ability to influence the VRM page replacement algorithm. Using this interface, AIX advises the VRM that certain pages should be purged from primary storage, that certain pages should be pinned in primary storage, or that previously pinned pages should be unpinned.

The AIX Operating System is based on UNIX System V with many enhancements. Among the enhancements are the facilities that utilize virtual memory. AIX runs in a virtual machine on the VRM.

AIX Kernel

The AIX kernel manages the various devices and resources that make up the virtual machine in which it resides. It is the control point for all virtual machine activity and the virtual machine resources.

The internals of the AIX kernel are modified and extended to allow it to run in a virtual machine, provide an extended process environment, and provide a usable and stable file system. The system call and subroutine interfaces allow many programs and utilities written for UNIX-compatible systems to run on AIX.



The kernel performs the following major functions:

- File System Management
  - File: open, close, read, write, change owner, get/change statistics, seek
  - File system: mount, umount, get statistics
  - Directory: change working directory, change root directory, make a directory, link to a file, unlink to a file
  - Security: access permissions.
- Process Management
  - Start and termination: fork a process, terminate this process, kill another process, kill a process group
  - Set process group
  - Informational: enable/disable accounting, get ID (process, parent, group), get times
  - Priority suggestion
  - Wait for child process to terminate
  - Lock data, text or stack in memory.
  - Signals: enable/disable signals, route signals to user routines, wait for a signal.
  - Semaphores: create semaphore, get semaphore ID, perform semaphore operations, delete semaphore.
- Memory Management
  - Private memory: grow, shrink
  - Shared memory: create, attach, delete.

- Time Management
  - Set time
  - Get time.
- Program Management
  - Execute a new program
  - Lock a program in memory.
- Resource Management
  - Set and get user and group IDs
  - Set and get user limits

Kernel Features

The kernel has the following features:

- Device Error Logging
- File System Enhancements
- Virtual Memory
- Enhanced Signals
- Customization Facilities.

The system command and utilities are programs that operate in unprivileged mode and use system calls to the kernel resources to perform functions. System calls to the operating system kernel are utilized to assist in the completion of the function or to actually perform the function. The system commands and utilities are divided into several categories based on the type of service performed:

User Access Control	Controls user access to the system.
System Status/Management	Provides system status.
Program Development	Provides program development aids.
Exchange Utilities	Provides exchange of files with other systems.
Migration Aids	Provides data interpretation between systems.
Information Handling	Provides data manipulation services.
Communications	Provides intra-system communications services.
Activity Monitoring and Accounting	Provides system trace and statistics.
Directory Management	Provides directory manipulation services.
File Management	Provides file manipulation services.
Queue Management	Provides queue manipulation services.

System Customization	Adds and deletes devices: • Changes device information • Displays configuration information.

The system commands and utilities are written in the C language, and the kernel is mostly written in the C language with some assembler language where necessary. The AIX kernel provides the virtual machine supervisory functions such as process management, file system management, input/output (I/O) control, and communication between processes and other miscellaneous facilities. Some of the AIX kernel functions are discussed in sections that follow.

Bootstrap

Before the kernel can run, it first must be loaded into segment 0 of the virtual machine. The bootstrap program is responsible for locating the kernel on the root file system, reading it into memory, and finally giving it control.

The bootstrap program resides on the minidisk adjacent to the root file system, but not a part of the file system. When a file system is created, the **mkfs** command handles putting the bootstrap program in the proper place on the minidisk. The **mkfs** command places the bootstrap program at the end of the minidisk on a block boundary. The file system uses the remaining blocks at the front of the minidisk. Block 0 of the minidisk, which is the virtual machine boot block, has a field that points to the bootstrap program, and a field specifying the length of the bootstrap program.

When the VRM is loaded and running, it is time to load the virtual machine (AIX). If a diskette is in any diskette drive, an attempt is made to load from the diskette. Otherwise, the minidisk directory is searched for a minidisk marked as an AIX root file system. When one is found, its boot block is read in order to find the bootstrap program. The bootstrap is loaded into the memory of the virtual machine at virtual address 0, and is given control.

The bootstrap program searches the root file system for the file **/unix**, reads its text and data segment into memory, and extends the segment. The bootstrap program moves itself out of the way, moves the kernel to start at address 0, and then gives the kernel control at its start entry point, thus completing the boot process. See "Creation and Execution" on page 1-16 for additional information.

Process Management

A ***process*** in the operating system is the current state of a program that is running. This includes a memory image (the logical layout of its parts in memory), the program text, program data, variables used, general register values, the status of opened files used, and current directory. Programs running in a process can be either operating system programs or user programs. A process must be active in order to request services to be performed by the kernel. Processes are paged into and out of memory when necessary. Processes not currently running are eligible to be paged from memory to disk.

User and Kernel Modes

The same process can be in either ***user*** mode or ***kernel*** mode. Normally, a user program while executing is called a user process and is considered to be in ***user mode***. When the process requires a function performed by the system, it calls the system as a subroutine. A process in user mode uses system calls to access system resources. This is also sometimes referred to as a kernel call. When the user process issues a system call, the environment switches from user to kernel mode. The system is running the same process. The difference is that the code running for the user process in this instance is kernel code. The process is now in ***kernel mode***. A process in kernel mode has full control of the system. When the kernel has completed the requested service, it usually returns control to the user mode of the process. A process in user mode can be preempted at any time. In contrast, a process in kernel mode usually cannot be preempted. Normally, a process in kernel mode runs until it voluntarily relinquishes control of the processor.

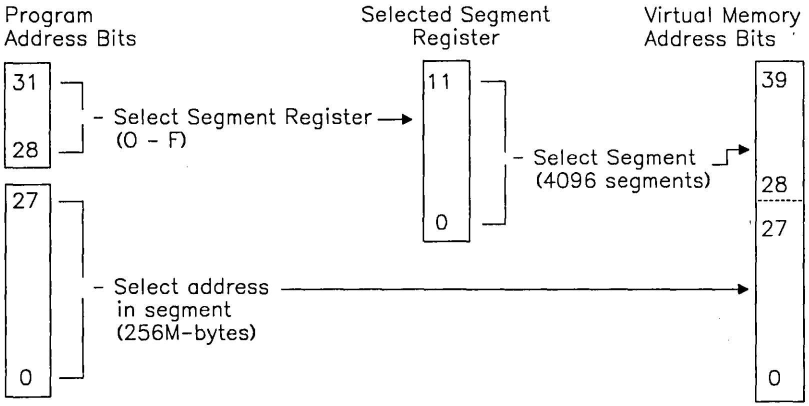
Several mechanisms can prompt a switch from user mode to kernel mode. One mechanism that causes a switch is the system timer. The system timer periodically interrupts the processor at fixed intervals per second. An interrupt is a signal that diverts the processor to a special software routine. During the service routine for the system timer, the kernel checks the priority of the processes for a possible change of process. The system scheduler performs the basic time-slicing to enable the processor to be shared among many users.

Servicing I/O requirements also causes a switch. Interrupt routines post completion of I/O operations. These routines start the next I/O operation on the device queue, mark all processes waiting for the service as ready to run, and set a flag to trigger a process switch when necessary upon return from kernel mode to user mode.

Memory Addressing

Memory management is provided to the operating system by the VRM (VRM). The VRM provides the operating system with paged virtual memory. Page faults can interrupt the operating system so that it can switch to some other task. Virtual memory functions are primarily controlled by SVCs from the operating system to the VRM, with interrupts used as appropriate.

Portions of a process can be addressed when a process is running in either user or kernel mode. The 32-bit virtual address space is divided into 16 segments. Each segment is 2^{28} bytes long. The segment registers provide access to the segmented virtual memory for the virtual machines. The virtual memory hardware allows a maximum of 16 concurrent segment accesses. The RT PC implementation restricts user mode processes to 14 concurrent segment accesses. A kernel mode process is permitted to concurrently access all segments accessible by the virtual machine. The segment registers provide several mechanisms for protecting the memory segments: by selective segment address loading into segment registers, and by page protection bit setting in each segment register. The protection settings provide a mechanism to invoke read and/or write protection in either machine state.


Program Address Bits
Selected Segment Register
Virtual Memory Address Bits
31
28
27
0
- Select Segment Register (0 - F)
- Select address in segment (256M-bytes)
11
0
- Select Segment (4096 segments)
39
28
27
0

Each segment register maps part or all of a logical RT PC program segment. All addresses are full 32-bit virtual addresses with the segment number occupying the leftmost 4 bits. The segment registers are a part of the process image and are therefore switched on each process dispatch. The kernel is mapped by segment register 0. This includes the kernel program text, data, and all I/O buffers. This mapping is fixed. The user program text segment is addressed by segment register 1 and the user program data segment is addressed by segment register 2. The user process stack and the user structure (**u.block**) are addressed by segment register 3. The user process stack grows from the high segment address to the lowest (downward). Segment registers 4 to 13 are used for the shared data manipulation in user programs. AIX provides a programming interface to manipulate these registers with the **shmget**, **shmat**, **shmdt**, and **shmctl** system calls.

The VRM reserves segment register 14 for direct memory access (DMA) and segment register 15 is reserved for bus I/O. This register is used to address the I/O communication channel (IOCC), floating-point adapter (FPA), and memory-mapped adapters. The following shows the run-time register assignments to the operating system.

Registers	Segments
0	Kernel: Text Data Stack
1	User Text
2	User Data
3	User Stack and User Block
4 ↓ 13	Shared Data
14	VRM-DMA
15	BUS

User Mode

A process in user mode accesses the following logical areas while running. These areas are used to store information.

Text segment — This segment is mapped by segment register 1 and is addressable by a process in user mode. The ***text*** segment occupies the low addresses in the virtual address space of a process. This segment usually contains the user program code that executes. The information in this segment originates from the load module that executed an **exec** system call. (The **exec** system call is briefly discussed later). During execution, this segment is read-only and a single copy of it is shared by all processes executing the same code.

Data segment — This segment is mapped by segment register 2 and addressable by a process in user mode. The ***data*** segment of a user process begins on the logical boundary above the text segment. The process has read and write access to this segment. This segment is not shared by other processes and its size can be extended using a **brk** system call. This segment contains an initialized portion used for data variables such as arrays, and a portion called ***bss***, which is initialized to zeros.

Stack segment — This segment is mapped by segment register 3 and is partially addressable by a process in user mode. This segment contains the user process stack and the user structure (**u.block**). The user structure is not addressable by a process in user mode. This segment of a user process starts at the high address in the process virtual address space and automatically grows in size toward the data segment as needed. This segment contains the run-time stack for a program and user programs can write to it. The process uses the top portion of this segment to pass I/O information to the kernel.

Shared segment

In addition to the text, data, and stack segments that each process uses, a process can create and/or attach itself to segments that are accessible by other processes. A set of system calls are available for using shared segments. When a shared segment is created or attached, the shared segment becomes part of the address space of the requesting process.

Shared segments can be used in either a read-only mode or in a read-write mode. Note that there is no implicit serialization support when two or more processes access the same shared segment. If one process reads from a particular area of a shared segment, then it is the responsibility of the two (or more) processes to coordinate their accesses to the shared area.

In addition to the sharing of segments, system calls are available that allow a process to logically superimpose the address range of a shared segment over an ordinary file in any mounted file system. Access to the file can then be made by accessing the segment. The segment can be shared with other processes or used by a signal process. There are three modes of mapping a file with a segment. They are read-write, read-only, and copy-on-write.

Read-write

A file mapped read-write allows loads and stores in the segment to behave like reads and writes to the corresponding file. If a process reads that portion of the segment that is beyond the logical end of file, the process will read zeros. If the process writes into that portion of the segment that is beyond the end of file, the file is extended.

Read-only

A file mapped read-only allows the file to be read. Any attempt to write to the file by storing into the segment will signal an error to the process. Just like read-write, a process that accesses the part of the segment that is beyond the end of file reads zeroes.

Copy-on-write

A file mapped copy-on-write also allows loads and stores to the segment to behave like reads and writes to the corresponding file except that the writes are temporary. That is, any storing into a copy-on-write segment modifies the segment but does not modify the corresponding file. The **fsync** system call writes the changed portions of the segment to the corresponding file, thereby making the mapped file an exact copy of the segment. If this system call is not issued, the file is never changed, allowing a process to cancel changes that it has made to a file if it decides the changes are not needed.

Kernel Mode

The following areas are addressable by a process in kernel mode. Except where noted, these areas are mapped by register 0. Data directly associated with a process are paged out of memory with the process. These areas contain all the data about a process needed by the kernel when the process is active. The four areas are:

Text This contains kernel program code that executes. It is read only by a user process.

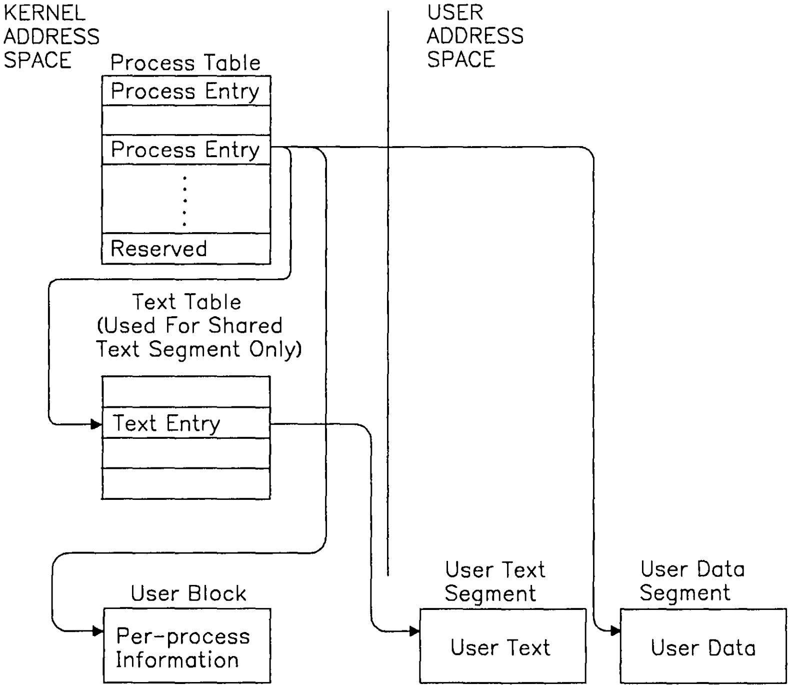
Global data This data can be addressed by any process while in kernel mode. It contains tables, such as the open file table and process table, and other data, such as buffer pointers, maintained by the kernel.

Per-process data This is sometimes called the **user structure, user area, u.area**, **user block**, or **u.block**. It is a portion of the user process stack segment. This area is paged with the process. It contains process information such as the current directory of files opened by the process or input and/or output (I/O) in kernel mode. This information occupies the top of the stack segment.

Stack This area is paged with the user-process. The kernel maintains a stack for each process. It saves the process information such as the call chain and local variables used by the kernel for the user process.

Process Data Structures

Most process management performed by the kernel is table searching and modification. The kernel maintains several tables to coordinate the running of many processes. The following figure shows the tables maintained by the kernel to manage processes.



The ***process table*** contains an entry for each process that is created. This table contains the data needed when the process is not running. The structure of this table can be found in the **/usr/include/sys/proc.h** file in the file system. This table is always in memory so the kernel can manage events for the process. Each table entry details the state of a process. The state information includes the segment IDs of the process, the identification number of the process, and the identification of the user running the process. There is one table entry for each process; therefore, the number of processes that can be created is determined by the size of the table, which is specified as a customize parameter, **procs** in the **/etc/master** file. Process creation causes an entry in the process table and process termination frees an entry in the table. One table entry is reserved for a process with **superuser** authority. A process is recognized as ***superuser*** process and is granted special privileges if its effective UID is 0.

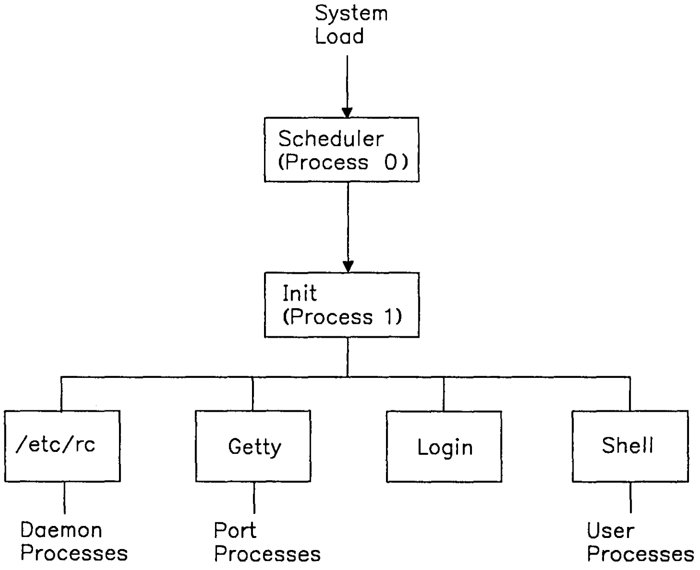
Each process has its own copy of the variable segments of the process, but the text segment can be shared. Sharing program text allows more effective use of memory. When text segments are shared by processes, the system maintains a ***text table***. This table is used to keep track of the shared text segment for each process sharing a text segment (a parent and child can share text after a fork, as an example). The structure of this table can be found at **/usr/include/sys/text.h** in the file system. A text table entry contains the segment ID of the text segment and the number of processes sharing this entry. When the number is reduced to 0, the entry is freed along with the segment. The first process executing a shared text segment causes a text table entry to be allocated and the segment to be created. A second process executing an already allocated text segment causes the number in the text table to be incremented.

The ***user structure*** (also called per-process data area or user block) contains information that must be accessible while the process executes. One user structure is allocated for each active process. The user structure is directly accessible to the kernel routines. This structure can be found at **/usr/include/sys/user.h** in the file system. This block contains information such as user and group identification numbers for determining file access privileges, pointers into the system file table for the files opened by the process, a pointer to the i-node table entry, and a list of responses for various signals. The user structure is part of the user stack segment. This chapter makes reference to entries in the user structure as **u.***xxxx*, where *xxxx* is the structure member.

The ***user data segment*** contains user data. The information consists of initialized data variables. A pointer to this segment is found in the process table entry. The user text segment contains program code. A pointer to this segment is found in the process table and if shared, the text table.

Creation and Execution

When the **/unix** file is found (see "Bootstrap" on page 1-9), it is loaded into segment 0 and executed. First, it initializes disk data structures such as the free-list blocks, I/O buffer pool, the pool of character buffers, and the list of available i-nodes.



After the initialization is complete, the kernel starts to build the first process (process 0), also known as the ***scheduler***. The scheduler is not created by the **fork** system call like other processes and it does not contain all the parts of a process. It is a unique process that contains only a data structure to be used by the kernel. Process 0 is the first entry in the process table and active only when the processor is in kernel mode.

Process 0 creates another process (process 1) by copying itself. Process 1 is also known as **init**. The system issues the equivalent of a **brk** system call to expand the size of process 1. Next, a program containing the instructions to perform an **exec** system call is copied into the text segment of the newly created process 1.

Process 0 is not a completed process image. The kernel will use this process for scheduling and controlling the operations of other system processes. Process 1 is the first completed process image and the ancestor of all subsequent processes. Neither process has run. The scheduler dispatches the first process that is ready to run. There is only one process ready to run, so process 1 runs. Process 1 immediately executes an **exec** system call to overlay itself with code from the **/etc/init** file.

As previously stated, all other processes are descendents of the **init** process. Normally, the **init** process runs the shell script, **/etc/rc**. The **rc** shell script is responsible for performing integrity checks, doing any necessary cleanup, mounting the normal file systems, enabling standard ports. After **/etc/rc** runs successfully, the **init** process creates a **getty** using the **fork** system call for each enabled port specified in the **/etc/ports** file. The **init** process performs the **exec** system call to **getty** to determine appropriate terminal speed and modes. The **getty** program performs the **exec** system call to **login** to validate password, sets the user ID (UID) and the group ID (GID), the current directory and so on. **login** execs shell or the program specified in the **/etc/passwd** file as the first program to be run after **login**. Shell runs in the same process created by **init**. Shell performs the **fork** system call, which creates new processes for every command. While the system is running, the **init** process sleeps waiting for the termination of any of its children. When a user logs off, **init** creates a new logger via a **fork**.

Parent and Child Processes

A process can, for various reasons, create a copy of itself. When this occurs, the original process is called the ***parent*** process and the newly created process is called the ***child*** process. The major difference between the original process, the parent, and the created process, the child, is that they have different process identification numbers, parent process identification numbers, and time accounting information.

The **fork** system call causes the total number of system processes to increase. A process uses the **fork** system call to create a copy of itself. The **fork** system call causes a new process, the child, to be created. Besides the differences mentioned previously, each receives a different value from the **fork** system call. (The child receives the value 0 and the parent receives the ID of the child process.) The two processes share open files and each process can determine whether it is the parent or the child by the value received. The parent may or may not wait for any of its children to terminate.

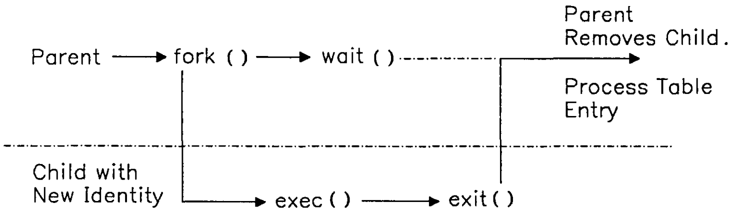
The **exec** system call causes the process to overlay the information it contains with new information. During an **exec** system call the process exchanges current text and data segments for new data and text segments. The total number of system processes does not change, only the process that issued the **exec** is affected. After the **exec** system call, the process identification number is the same and open files remain open (except close-on-exec files).

The **exit** system call terminates the process that issued the **exit**. All files accessed by that process are closed and the waiting parent is notified. A ***zombie*** process is a terminated process whose entry remains in the process table. The parent process is responsible to clear the entry from the process table. In the case of a child whose parent has terminated, **init** becomes the parent process clears the entry. If accounting is enabled, **exit** writes an accounting record.

The **wait** system call suspends the calling process until the child process **exits**, the child stops in trace mode (the child is traced by its parent), or the caller receives a signal. A **wait** system call passes termination status to the parent process, 1 byte (high) passed by **exit** and 1 byte (low) of system status. This system call also removes zombies from the process table.

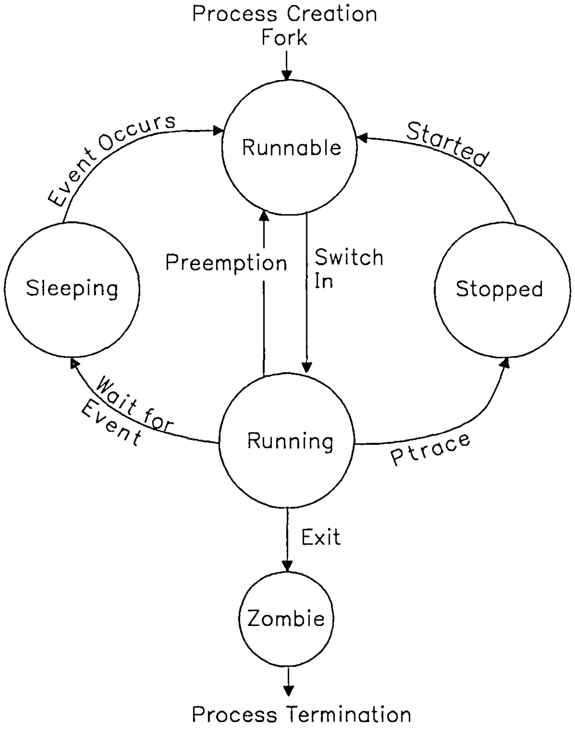
The following scenario discusses a parent process and child process relationship and the system calls to synchronize them. It is important to note that the parent process may terminate before the child process. In this instance, the **init** process assumes the role of the parent process.

A parent process executes a **fork** system call, producing a new process. The new process executes an **exec** system call creating a child process with a new identity. This is similar to the sequence **shell** uses when it runs a program. The **wait** system call causes a parent process to wait for the child to finish processing. When running interactively, the **shell** process executes a **fork** system call, the child process (shell running in the new process) executes an **exec** system call for the required program, and the parent process (shell) executes a **wait** system call to wait for the child to finish running. When the child executes an **exit** system call, the parent causes the process table entry for the child to be removed and prompts for another command. When running in the background, the **shell** process simply prints the process ID of the child and does not wait for the child process to terminate. See the following for the relationship of the parent and child processes as described when they run interactively.



States of a Process

A process can be in one of many states. A process can be ready to run, running, sleeping (waiting on an event), stopped, or ended. The scheduler determines which order the competing processes execute. The following diagram shows the process states and the events that change the states.



Only one user process is active or running at any given time. All other user processes are suspended from running. For example, a process that is waiting for any of its children to end, waits for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by the process table entry of its parent. When the event occurs, the process is awakened. When a process is awakened, it is ready to run, which means it is eligible to be dispatched. Normally, processes run to completion unless they sleep. They sleep for reasons such as waiting for input or output to complete, time slices, waiting for an event to occur or signals from other processes. At each timer interrupt, the timer interrupt routine examines the process queues, and may cause a process switch. When a process is sleeping, it may be paged out of memory. The process switch routine will not restart a process that is paged out. It checks that kernel and user data for a process are addressable before it restarts the process.

A process that relinquishes control of the processor is usually waiting for some I/O to be performed. In that case, the process issues a **sleep** call specifying **chan**, which is usually the address of the kernel data structure, and specifies a wakeup priority. It normally remains in a sleep state until a **wakeup** call is issued specifying the same **chan**. If the wakeup priority is low enough for the signal to be processed, the process is awakened and restarted in the same mode prior to sleep. Sometimes many processes may be waiting on the same event to occur, such as memory allocation. Since this is possible, when the process returns from sleep, it must first check that the event or resource was not seized by another process waiting on the same **chan**. If the resource is not available, the process issues another **sleep** call.

Priority Computation

Each process has an assigned priority. User processes are assigned low priorities. The scheduler uses the process priorities to dispatch processes. It dynamically calculates process priorities to select the inactive, but ready to run, process to run when the currently active process stops. A system process has a higher priority than any user process.

User process priorities are assigned by an algorithm based on the ratio of the amount of compute time to real time recently used by the process. At every tick of the system timer, the **p_cpu** field (processor usage) in the process table for the running process is incremented. The compute time to real-time ratio is updated every second. Using negative exponential distribution, the kernel decreases **p_cpu** by half its value for every process at or above the base user level and recalculates the priority of the processes. Processes that accumulated a lot of execution time have less priority than processes with very little execution time. A user process can execute a **nice** system call to induce a bias in the calculation. Ordinary user processes can only decrease their priority, while root user processes can either increase or decrease their priority.

Signals

Signals provide communication to an active process, forcing a single set of events where the current process environment is saved, and a new one is generated. A process can designate a signal handler function to respond to the signal. The signals all have the same priority, and critical functions can protect themselves from signal interference.

A signal is an event that interrupts the normal execution of a process. The set of signals is defined by the AIX system, and they are listed in the discussion of "signal" on page 2-145. All signals have the same priority.

A process can specify a ***signal handler*** subroutine, which is to be called when a signal occurs. It can also specify that a signal is to be ***blocked*** or ***ignored***, or that a ***default action*** is to be taken by the system when a signal occurs.

A global ***signal mask*** defines the set of signals currently blocked from delivery to a process. The signal mask for a process is initialized from that of its parent. It can be changed with a **sigblock** or **sigsetmask** system call. While a signal handler is executing for a given signal, the signal that caused it to be called is blocked, but other signals can occur. When the handler finishes, the signal is again unblocked.

Normally, signal handlers execute on the current stack of the process. This can be changed, on a per-signal basis, so that signal handlers execute on a special ***signal stack***.

When a signal is sent to a process, it is added to a set of signals pending for the process. If the signal is not currently blocked, then it is delivered to the process. When a signal is delivered, the following actions occur:

1. The current state of the process execution context is saved.
2. A new signal mask is calculated, which remains in effect for the duration of the process's signal handler or until a **sigblock** or **sigsetmask** system call is made. The new mask is formed by logically OR-ing the current signal mask, the signal being delivered, and the signal mask associated with the handler to be called.
3. If the signal handler is to execute on the signal stack, then the current stack is changed to the signal stack.
4. The signal handler is called. The parameters that are passed to the handler are defined in the following description.

   The signal-handler subroutine can be declared as follows:

   > *handler* **(***sig***,** *code***,** *scp***)**
   > **int** *sig***,** *code***;**
   > **struct sigcontext ****scp***;**

   The *sig* parameter is the signal number. The *code* parameter is provided only for compatibility with other UNIX-compatible systems, and its value is always 0. The *scp* parameter points to the **sigcontext** structure that is later used to restore the process's previous execution context. The **sigcontext** structure is defined in **signal.h**.
5. If the signal-handling routine returns normally, then the previous context is restored and the process resumes at the point at which it was interrupted. The handler can cause the process to resume in a different context by calling the **longjmp** subroutine. (For information on how to save and restore the execution context, see "setjmp, longjmp" on page 3-332.)

After a **fork** system call, the child process inherits all signals, the signal mask, and the signal stack from its parent.

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so. Ignored signals continue to be ignored, the signal mask remains the same, and the signal stack state is reset.

When the **longjmp** subroutine is called, the process leaves the signal stack, if it is currently on it, and restores the signal mask to the state when the corresponding **setjmp** call was made. See "sigvec" on page 2-156 for enhanced signal information.

The operating system has five signal classes:

- Hardware signals occur as the result of conditions such as arithmetic exceptions, illegal instruction execution, or memory protection violations.
- Software signals are generally user-initiated interrupts. Termination, quit, and kill are signal types that represent various levels of user or program-initiated signals to a process. In addition, timer expiration can be signalled with software-driven alarm signals.
- A process can be notified of an event that occurred based on some descriptor, or nonblocking operation that completes. A process can also request a catastrophic condition signal.
- Processes can be stopped, restarted, or can receive notification of state changes in a child process.
- Processes can receive threshold warnings when the processing unit time limit or a file size limit is reached.

The kernel also contains additions and modifications to enhance the unsolicited interrupt signal system for kernel-to-process communications.

File System Management

Files within the file system are grouped into directories and the directories are organized into a hierarchy. At the top of the hierarchy is a directory called the root directory. This directory is designated as / (slash). The root directory contains some system-related files and standard directories such as **/bin, /usr, /dev, /etc,** and **/lib**. Files can be attached anywhere onto the hierarchy of directories.

A file is a one-dimensional array of bytes uniquely identified by a device name (major and minor number) and an **i-number** (index number). Data within the file is located in blocks. The logical blocks in the AIX file system are 2048 bytes long. Block size is specified in the superblock; therefore a compatible file system can be mounted and used on the AIX system. Block size of the mounted file system is recognized by the kernel.

Types of Files

AIX file system files can be directory files, ordinary files, or special files. All files have read, write, and execute permissions for the owner, group, and others. The read, write and execute permissions on a file are granted to a process if one or more of the following are true:

- The effective UID of the process is superuser authority.
- The effective UID of the process matches the UID of the owner of the file and the appropriate access bit of the *owner* portion (700 octal or 0x01c0) of the file mode is set.
- The effective UID of the process does not match the UID of the owner of the file, and the effective GID of the process or one of the process or one of the GIDs in the multiple group list of the process matches the group of the file and the appropriate access bit of the *group* portion (070 octal or 0x38) of the file mode is set.
- The effective UID of the process does not match the UID of the owner of the file, and the effective GID of the process does not match the GID of the file, and the appropriate access bit of the *other* portion (007 octal or 0x07) of the file mode is set.

Otherwise, the corresponding permissions are denied. When a process creates a file, the GID of the file is the effective GID of the process.

In a directory, however, the read, write, and execute permissions are interpreted differently from ordinary files. Read permission for a directory indicates that standard utility programs are allowed to open and read the information in the directory. Write permission for a directory indicates that files in the directory can be created or removed. Execute permission for a directory indicates that a user can search the directory for a file name. Denying search privileges for a directory provides protection against using files in that directory. A request to changing location into a directory where execute permission is denied cannot be performed.

AIX permits file names to be up to 14 characters long. It is possible for one file to have several names. Any printable character can be used in the name. Names containing unprintable characters, space characters, tabs, and shell metacharacters are not recommended. The AIX file system reserves file names **.** (dot) and **..** (dot dot); therefore these names cannot be used as file names.

Directory Files

The AIX file system hierarchy centers around directory files. Directory files contain lists of files. The AIX operating system maintains the directory files. Executing programs can read the directory files, but AIX prevents programs from changing directory files to protect the information in the directory files. Programs may add entries to directories by requesting the system to create a file. The system is responsible for making the changes to directory files. Files listed in a directory can be ordinary files, directory files, or special files.

Ordinary Files

Ordinary files are attached to directories. An ordinary file might contain an executable program, document text, or other types of information that can be processed. There are two types of ordinary files: text files and binary files. Text files normally contain ASCII (American Standard Code for Information Interchange) characters. Binary files contain 256 possible values for each byte.

Special files

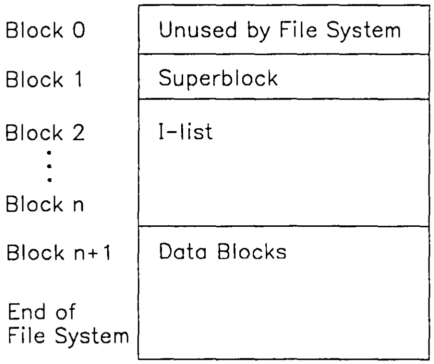
Special files are used to provide a convenient channel for accessing input and output (I/O) mechanisms to devices. For each I/O device, including memory, there is a special file. Most special files are found in the **/dev** directory. Special files provide an interface between application programs and the AIX kernel routines dealing with the devices. The names of the special files indicate the type of devices with which they are associated. Special files are read and written just like ordinary files, except read and write requests activate the associated device.

There are two types of special files: character and block. Some devices, such as a terminal, handle one character at a time. The character special files provide access to character I/O devices. Some I/O devices, such as a disk, transfer data in blocks at a time for efficiency. The block special files provide access to block I/O devices.

No characters are stored in a special file. When a directory that contains special files is listed, it identifies major and minor device numbers associated with the device rather than file length. The major device number identifies the type of I/O device that the file references. The minor device identifies the specific device when multiple devices of the same type exist, such as terminals.

File System Layout

For this discussion, the device that contains the file system is a minidisk with logical data blocks of 2048 bytes. Thus, a unit of disk storage or block is 2048 bytes. Blocks are numbered sequentially from the beginning of the minidisk, starting with 0. A file system is logically separated into four sections as shown in the following figure.



Block 0

The file system does not use block 0. This block usually contains system bootstrap information.

Superblock

Block 1 is the ***superblock***. See "fs" on page 4-74 for a detailed description of the contents of this structure. This block is used to keep track of the file system. Some of the file system information contained in the superblock is:

- File system size in logical blocks
- File system name
- Number of blocks reserved for i-nodes
- The i-node list
- The free-block list.

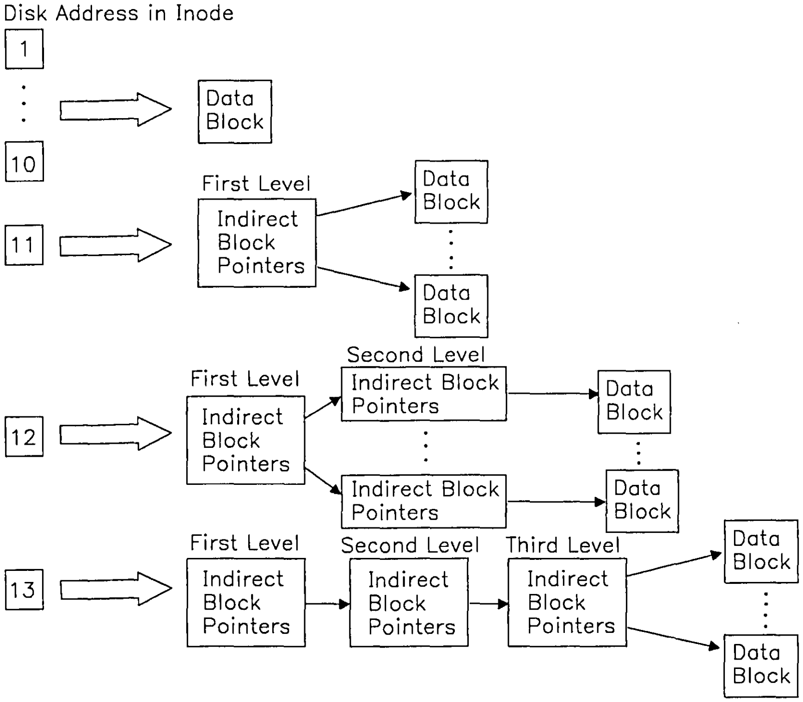
I-list

Blocks 2 through *n* are the ***i-list***, which contain structures relating a file to the data blocks on disk. The size of this section depends on the size of the mounted file system. Each structure, called an ***i-node***, is 64 bytes long. Each i-node designates a file. See "inode" on page 4-92 for the detailed content of the i-node structure of an ordinary file or directory. Each i-node structure is sequentially numbered from 0 to a maximum number, which is dependent on the file system size. Each index number, or ***i-number***, designates a 64-byte i-node and is used as an offset within the i-list. I-number 1, the first 64 bytes, is not allocated by the file system. Usually i-number 2 is the i-node of the root directory. The remaining i-numbers are allocated by the file system. The i-node contains information about each file such as:

- Mode and type of file
- Length of file
- ID numbers of owner and group
- Relevant dates and times
- Number of links
- Location of file blocks.

I-node Addresses

An i-node contains thirteen 4-byte disk addresses. The following figure shows the use of the disk addresses in the i-node.



Addresses 1 through 10 point directly to the first 10 disk blocks in the file. Addresses in indirect blocks are 4 bytes long. If the file is larger than 10 blocks, address 11 points to a first level indirect block containing the next **blocksize**÷**4** block addresses in the file. This is called indirect addressing. Depending on the size of a block, an indirect block contains blocksize÷4 addresses. For example, the indirect block for a file system on diskette contains 512 ÷ 4 or 128 addresses; a disk indirect block contains 2048 ÷ 4 or 512 addresses. A larger file requires use of the address 12. This address points to a second-level indirect block, which contains the addresses of up to blocksize ÷ 4 first-level indirect blocks. If the file is larger, address 13 is required. This address points to a third-level indirect block, which contains the addresses of up to blocksize ÷ 4 second-level indirect blocks. Any of these addresses can be 0, indicating holes in the file, which are read as binary zeros. Indirect block numbers can be 0 when the file contains large holes.

I-number Allocation

The file system tracks free i-numbers that are available. It maintains a list of i-nodes available for allocation in the superblock. The superblock contains the following information to allocate i-numbers.

- s_**inode**, an array containing the next free i-numbers to be allocated to files.
- s_**ninode**, the count of free i-numbers in the array. This is used as an index into the s_**inode** array.
- s_**tinode**, the total number of i-nodes in the file system.

Allocating an i-number to a file when **s_ninode** is greater than 0, **s_ninode** is decremented to get the next available i-number from **s_inode**. If **s_inode**[*s_ninode*] is 0, the next free i-numbers available from the i-list are placed onto the array and another attempt to allocate is made. Freeing an i-number when **s_ninode**[*s_ninode*] is less than maximum, places the freed i-number into the array and the count increments.

Data Blocks

The last section of the file system is data blocks, which contains data stored in files, indirect blocks that point to other data or indirect blocks for large files, or blocks that are available for data. These blocks are 2048 bytes long. The i-node contains the addresses of the data blocks that are already used in files. Otherwise, the data blocks are free and available for allocation to a file.

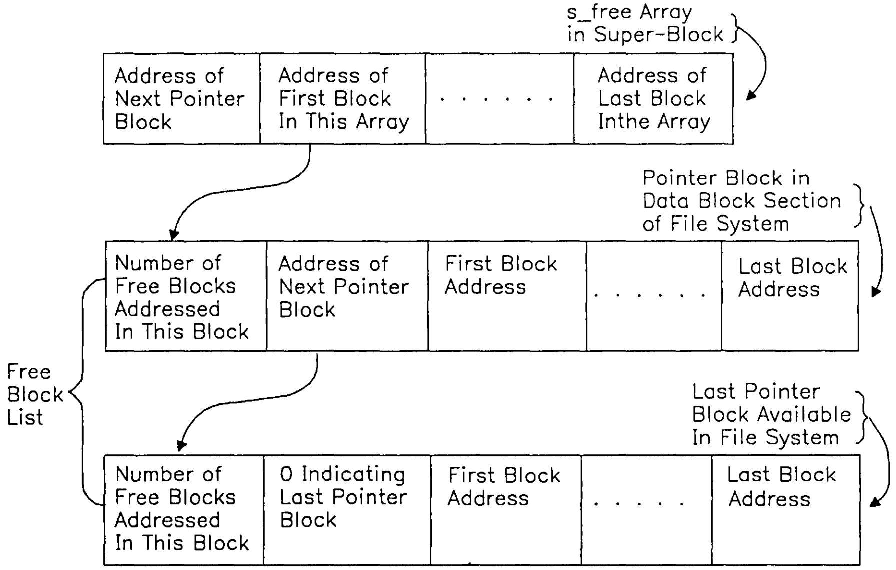
Free-block List

The file system maintains a list of all free blocks in a **free-block list**. The free-block list is a linked list of pointer blocks. A free block is a block that is not allocated to the superblock, i-nodes, indirect blocks, or files. Blocks are allocated dynamically to a file when needed from the data block section of the file system. In order to track data-block allocation, the superblock contains the following:

- **s_free**, an array of free block addresses.
- **s_nfree**, the number of free blocks in the **s_free** array. This is used as an index into the **s_free** array.
- **s_tfree**, the total number of free blocks available in the file system.

Allocating Blocks

Each pointer block in the free-block list contains a count of the number of entries in the block, up to 50, and the address of the next pointer block. If the pointer has a value of 0, this indicates the last pointer block in the file system. The first long integer in each pointer block is the number, up to 50, of free blocks addressed in the block. The next long integer is the address of the next pointer block available. The next 49 long integers contain the addresses of 49 free blocks. The following figure shows the relationship of the free-block list and **s_free** array.



s_free Array in Super-Block
Address of Next Pointer Block
Address of First Block In This Array
. . . .
Address of Last Block Inthe Array
Pointer Block in Data Block Section of File System
Number of Free Blocks Addressed In This Block
Address of Next Pointer Block
First Block Address
. . . .
Last Block Address
Free Block List
Last Pointer Block Available In File System
Number of Free Blocks Addressed In This Block
0 Indicating Last Pointer Block
First Block Address
. . . . . .
Last Block Address

The file system allocates free blocks using the **s_free** array and pointers in the superblock. The pointer block information is copied into the **s_free** array and superblock as follows. **s_free**[*0*] contains the address of the next pointer block in the free-block list. The remainder of the **s_free** array contains addresses of the free blocks in this pointer block. **s_free**[*s_nfree-1*] contains the address of the next free block available to be allocated.

Allocating a block causes **s_nfree** to be decremented to locate the next available block. If decrementing **s_nfree** caused its value to become 0, this indicates that more blocks are not available in the **s_free** array. Therefore, the address found is the location of the next pointer block. File system management reads the pointer block into the superblock, placing its first long integer in **s_nfree** and copying the next 50 long integers, which are addresses, into the **s_free** array. If the location of the next free block is 0, indicating the end of the chain, then new blocks are not available in the file system. This indicates an error condition.

When a block is freed and **s_nfree** has a value less than 50, the new block address is added to the **s_free** array and **s_nfree** is incremented. When a block is freed and **s_nfree** equals 50, **s_nfree** and the **s_free** array are copied into the freed block and the value **s_nfree** becomes 0. The file system manager updates the **s_free** array as previously described when **s_nfree** was equal to 0.

Directory Contents

A directory file is like an ordinary file except that it cannot be written by a user. See "dir" on page 4-60 for a detailed description of a directory. A bit in the i-node identifies the file as a directory file. The directory contains an entry for each file or subdirectory within it. A directory block contains 16-byte directory entries for each file within it. The first 2 bytes of each entry are the **i-number** for the file. The remaining 14 bytes are the file name, which is left adjusted in the field. The first file entry in a directory is its i-node and entry **.** (dot). The second file entry is the parent directory, **..** (dot dot). The third entry designates the first actual file in the directory.

Path Name Resolution

A path name through the file system is a route of directories and i-nodes. A direct path starts at the file system root. A relative path starts at the current directory. The last name in the path references a file. The following example shows the resolution of a direct path; the resolution of a relative path is similar.

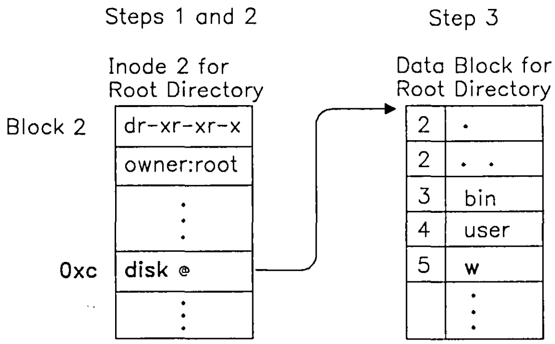
Direct Path

The following describes accessing a file with a direct path name. Consider the path name **/w/z**. This path starts at the root directory. It leads from the root directory to the directory **w** and then to the file **z**.

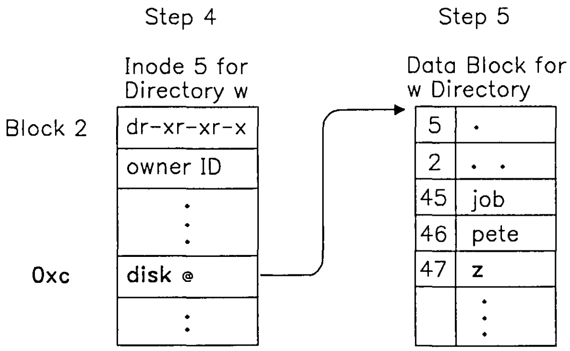
1. Read address 1 of i-number 2 (root) for the address of the root directory.

2. Read the i-node for the root directory.

3. Use the information in the root directory to search the root directory for the name **w** and its i-number.



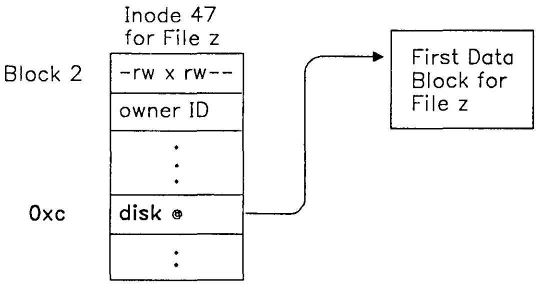
4. Read the i-number for **w**.



5. Use the information in the **w** i-node to search the **w** directory for the file named **z** and its i-number.

6. Read the i-node for **z**. The addresses for the file blocks assigned to the files start at offset 10 within the i-node. The first 10 are direct addresses as previously described; the last three are indirect.

Step 6



Relative Path

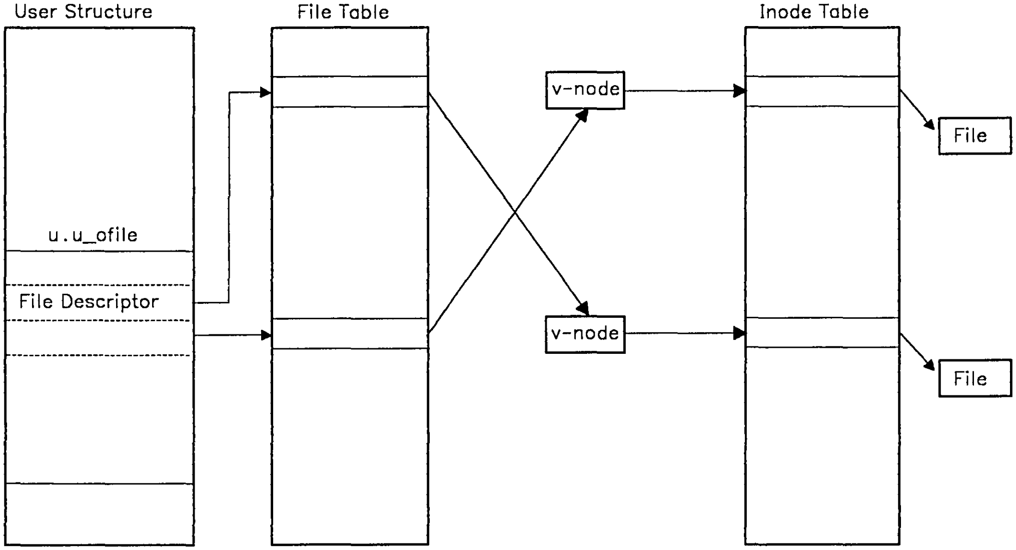
Consider the relative path name **..(dot dot)/x/y**. The path leads from the current directory, to the parent of the current directory, to the parent subdirectory **x**, and finally to the file named **y** in the directory **x**. In order to follow this path, the system performs the following steps:

1. Read the i-node of the current directory.
2. Use the information in the i-node for the current directory to search the current directory for the name **..** (dot dot) and its i-number.
3. Read the i-number for **..** (dot dot).
4. Use the information in the **..** (dot dot) i-node to search the parent directory for the file named **x** and its i-number.
5. Read the i-node for **x**.
6. Use the information in the **x** i-node to search the **x** directory for the file named **y** and its i-number.
7. Read the i-node for **y**. The addresses for the file blocks assigned to this file starts at offset 10 within the i-node. The first 10 are direct addresses as previously described; the last three are indirect.

File System Data Structures

The kernel maintains structures in memory, along with the superblock, to access files in the file system. These structures include the v-nodes, i-node table, and file table.

Access to the files in the system begins at the per-process data region in a process (user structure). System calls provide access to file system services for user processes. The most common functions performed are **open, creat, read, write, lseek,** and **close**. The user structure contains an array, **u.u_ofile**, that is indexed by file descriptor values. This array of entries contains the addresses of the file table entries for each file opened or created by the process and used for I/O operations. Descendants of the process inherit the contents of the **u.u_ofile** array. The following figure shows the data structure relationships for accessing two files. The format of the user structure is found in **/usr/include/sys/user.h**.



All operations on files are performed with the aid of the corresponding i-node table entry. When the system accesses a file, it locates the corresponding i-node (see "Path Name Resolution" on page 1-30), allocates an entry in the i-node table, reads the i-node into memory, and creates the corresponding v-node. The entry in the i-node table is the current version of the i-node and is the focus of file system activity. The structure of an i-node entry is found in **/usr/include/sys/inode.h**. The i-node table contains the key information for accessing a file including flags, owner, mode, mounted-on device, i-number, and location of file blocks.

The v-node holds information on the operations that the rest of AIX uses to perform file-related system calls on the corresponding i-node and file. These operations can differ from file to file. For example, a file residing on a remote server is accessible through a set of Distributed Services operations when Distributed Services is installed. The structure of the v-nodes is found in **/usr/include/sys/vnode.h**.

Another table the kernel maintains in memory for accessing files is the **file** table. The structure of a file table entry is found in the **/usr/include/sys/file.h** file in the file system. A file table entry is associated with **open** and **creat** calls for each file. Each entry in the file table contains the read/write offset of the file and a pointer to a v-node, which in turn contains a pointer in its private data field to a particular entry in the i-node table. The user process maintains a file table entry for each file it opened or created. After a fork, the two processes share the file table entries. A separate open of a file that is already open shares the i-node table entry but has distinct file table entries.

I/O Control

The kernel and user processes use calls to the system to access the I/O subsystem. System calls that perform I/O usually cause the calling process to be suspended (it relinquishes control of the hardware processor) while the I/O is being performed. Another process that is ready to run is dispatched.

The supervisor call (SVC) is the mechanism that permits a virtual machine operating system to request services from the VRM. The AIX kernel sends requests to the hardware devices using supervisor calls to the VRM through the Virtual Machine Interface (VMI).

Each physical device that is attached to the system must communicate with the AIX kernel via a device driver. Due to the VRM/VMI boundary, the AIX device drivers do not deal directly with actual device interrupts and do not directly place requests on the hardware bus. Each AIX device driver that deals with a physical device has a corresponding part that handles the physical device in the VRM. AIX device drivers communicate with their VRM counterpart using supervisor calls directed to the VRM.

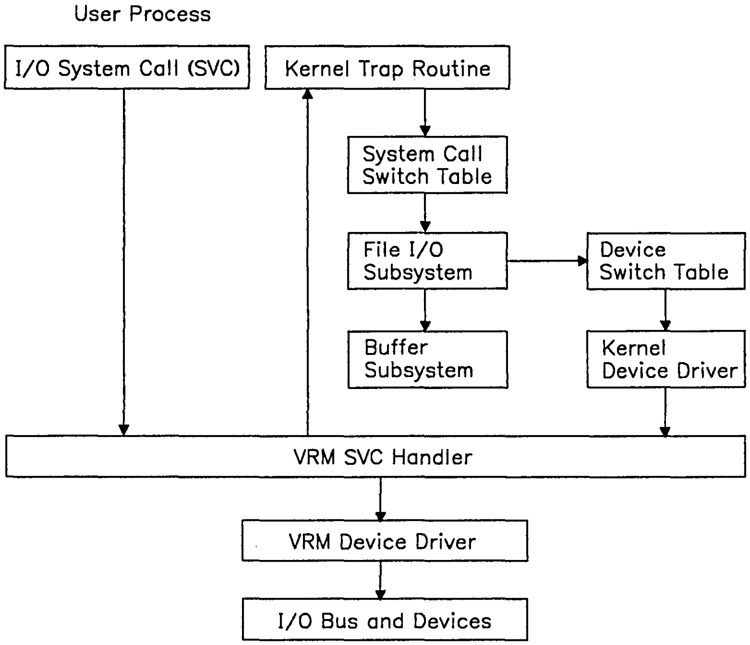
Each AIX device driver is activated by a basic I/O system call (open, read, write, ioctl, close, and so on) from the application level. AIX system calls are invoked via SVCs that are distinguished from VRM SVC requests by the SVC number. There is a range of SVC numbers (0 - 32767) reserved for virtual machines and a different range of SVC numbers (32768 - 65535) used for requests to the VRM.

The AIX device driver routine for the particular device class performs the requested I/O function to a device by issuing VRM SVC requests. There are two basic device classes, one for block-oriented devices, and one for character-oriented devices.

Each AIX device driver is addressed by individual applications via a system call (SVC) interface. The application builds device-dependent commands and data streams, and invokes the appropriate AIX device driver. To accomplish the system call, the device driver maps the system call inputs into VRM I/O subsystem SVCs.

The AIX device drivers perform basic device error determination by reading device status to determine exception conditions. In some instances (where the device driver is driven with an **ioctl** system call), specific commands are *passed through* to the VRM device driver by the device driver. In general however, the AIX device drivers package the commands and data for the VMI I/O subsystem interface and execute the SVC calls across the VMI. Virtual interrupts across the VRM are serviced to satisfy each outstanding system request.

AIX device drivers are explained later in this discussion. See *Virtual Resource Manager Technical Reference* for an explanation of VRM device drivers. The following illustration shows an overview of the relationship in the control flow of the I/O subsystem.



Kernel Trap Routine

Each system call is interpreted as a request to perform a predetermined function. The function to be performed is determined by a trap handler in the kernel. This kernel trap routine is called in other instances besides system call handling. This routine also runs in cases of error conditions or interrupt handling.

During a system call, any error indicators are reset and the process return status is saved. Next, the system call is used to determine a system call number. (An integer number is assigned for each type of system call.)

System Call Switch Table

The system call number is used as an index into the system call switch table. This table contains the address for the specific handler routine that handles the call. A call is made to the system call handler routine, which receives the parameters supplied by the user program along with the system call. This routine copies the parameters out of the user part of the process to the kernel part of the process.

File I/O Subsystem

The system call switch table contains many entry points into the file I/O subsystem. Common entry points used are **open, close, read, write, lseek** and **ioctl** system calls. The file I/O subsystem determines whether the system call is to gain access to an ordinary file, a block special file, or a character special file. In the case of special files, this subsystem translates the file name into a major and minor number, which is used to select the device and/or routine.

Buffer Subsystem

The buffer subsystem maintains a system buffer pool that is used by block devices to read and write data. Requests for blocks found in the pool are returned immediately to the requester. If blocks are not found in the pool, the least recently used (LRU) buffer is freed and allocated. See"Block Device Drivers" on page C-15 for more details.

Device Switch Table

The device switch table is used as an interface to the device drivers. The device driver ***major*** number is used to select the proper routine. The ***minor*** number selects one of multiple subdevices. See "Device Management" on page 1-39 for more details about device drivers.

Kernel Device Driver

The device driver in the kernel does not issue I/O directly to the device. Instead, it issues an SVC to a device driver in the VRM to perform the actual I/O. When the VRM device driver has completed its task, it sends a virtual interrupt to the virtual machine.

VRM Device Driver

The VRM device driver accesses the hardware device by accessing the memory mapped I/O bus.

I/O Bus

The I/O bus is addressed via segment register 15 and is accessed like ordinary memory. Normally, only VRM device drivers should attempt addressing the I/O bus. However, a kernel device driver can address the I/O bus if proper device handling has been done. Normally, the bus address space is protected from access by opening the bus special file. See "bus" on page 6-5 for additional information.

Return

When the routine returns, the return status is copied back into the user part of the process and the process resumes running. Some rescheduling of processes can occur upon return from kernel mode due to interrupts or errors while processing the system call. Execution starts in the program immediately after the system call unless an error occurs.

Common Routines

Kernel and user processes use calls to kernel routines as an interface to the I/O subsystem. These routines must prepare the system internal tables in order to ensure proper performance. These routines are invoked using system calls. The following describes the common routines used and their effects on tables maintained by the operating system.

Creat and Open

The **creat** and **open** routines create and/or open a file for reading or writing and return a file descriptor for the opened file. First, the file system directory is scanned to locate the named file. An i-node is created if not found and an entry is placed in an **i-node** table. This entry is somewhat different than the i-node as it exists on the disk. It contains a count of the users (used by **close**) and disk block addresses are expanded from the 3 bytes stored in an i-node to the minidisk block number. There is one i-node table entry for a given file. An i-node table entry exists for an open file, the current directory of a process, or a special file containing a currently mounted file system.

The open file causes a **u.ofile** array to be stored in the **user structure**. The **read**, **write** or any other routines that perform operations on the opened file use the file descriptor returned as an index into this array. Array entries are pointers to corresponding entries in the **file** table that is maintained by the system.

Each **creat** or **open** of a file causes one file table entry to be created. If a file is opened by more than one process, this table contains multiple entries. After a process performs a **fork** system call, the resulting processes share the same entry of the opened file in the table. The **fork** system call increments the reference count entry in the table. This count is used by **close** to determine when the entry can be removed from the table. Additionally, it contains a pointer to a v-node, which in turn points to the entry for the file in the i-node table.

Close

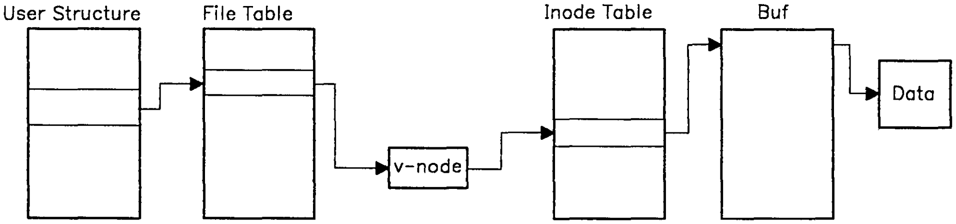
The **close** routine is called each time a process closes a file. When the last process closes the file, the i-node table entry is removed. In some instances, buffers containing data for the file that are queued but not written, are written to the file before the **close** completes.

Read and Write

The **read** and **write** routines use parameters supplied by the user and the file table entry to set the variables **u.u_base, u.u_count,** and **u.u_offset** (in the **user** structure). These variables contain the user address of the I/O target area, the byte count for the transfer, and the current location within the file. It may be necessary to transform the current location into a logical block number or physical block number depending on the target.

I/O Data Structures

The operating system maintains data structures to track I/O processing to and from devices. The following figure shows these data structures and their relative relationship.



When an **open** or **creat** occurs, an entry is made in the file table. This table is referenced by pointers from the **user structure** using file reference numbers passed to system calls.

Another data structure is the **i-node table**, which contains one entry for each active i-node. Each entry maintains an open count and a link count, which is used by **close**. This table is referenced by device number and i-number. The entry in this table is created by the **open** routine and removed by the **close** routine (when the open count and link count are 0). The i-node table array of a file is found by following a series of pointers. The first pointer is in the user structure, which points to a file table entry, which points to a v-node, which points to the i-node table entry. (See above figure.)

The **user structure** contains information accessed by the user process, kernel, and the device driver routines to perform device I/O requests. The elements of this block are needed when performing I/O.

Buf is a table of buffer headers maintained by the kernel and used for data read from or written to block devices. Each buffer header has at least three parts: flags (to show status information), forward and backward pointers (to maintain two doubly linked lists; the **b** list to link the buffers that correspond to a disk block and the **av** list to link buffers that are available for reuse). The structure of the buffer header can be found in location **/usr/include/sys/buf.h** in the file system.

Device Management

The operating system uses ***special files***, sometimes called ***device files***, to refer to specific hardware devices and device drivers. Special files, at first glance, appear to be files just like any other. They have path names that appear in a directory, and they have the same access protection as ordinary files. They can be used in almost every way that ordinary files can be used. However, an ordinary file is a logical grouping of data recorded on disk, but a special file corresponds to a device (such as a line printer), a logical subdevice (such as a large section of disk drive), or a pseudo-device (such as the physical memory of the computer, **/dev/mem**, or the null file, **/dev/null**). By convention, all special files supplied with AIX are located in the **/dev** directory.

Device Drivers

A ***device driver*** is a set of routines that are installed as part of the AIX kernel to control the transmission of data to and from a device. The major interface between the kernel and the device drivers is through the device switch table.

Major Device Number

A major device number designates which device driver in the operating system is to handle I/O requests. The major device number for each device is assigned in the **/etc/master** file, which is used in system configuration (see the **config** command in *AIX Operating System Commands Reference*).

Minor Device Number

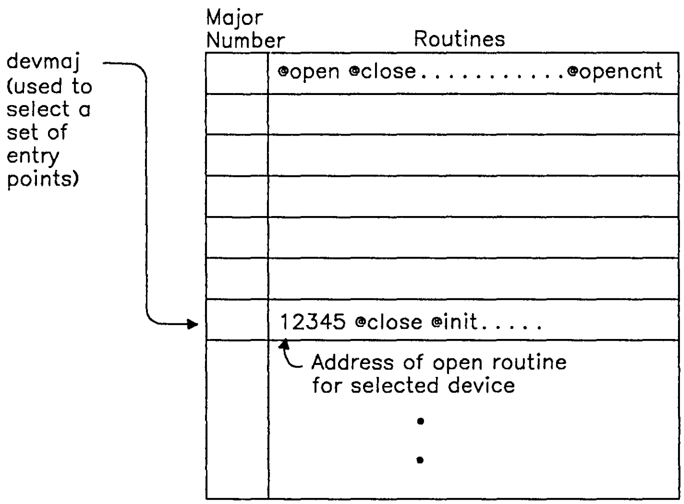
The interpretation of the minor device number is entirely dependent on the particular device driver. The minor device number is frequently used to index an array that contains information about each of several virtual devices or subdevices. For each virtual device, there exists an I/O device number (IODN) that is passed on all I/O supervisor calls. This IODN is used by the VRM SVC handler to route the I/O request to the proper VRM device driver. This IODN is either a fixed assignment or can be dynamically assigned by the **config** program. See **config** in *AIX Operating System Commands Reference.*

Requests for Device I/O

The operating system controls the processing of all user I/O requests and device interrupts. When a user program requests I/O to a device using system calls, control is transferred to the kernel. If the system call is to a device (not an ordinary file), the path pointer points to a special file. Special files describe the device and indicate to the system that the call is for a device. If the requested file is a special file, the system records the major and minor device numbers in the i-node table entry.

All devices attached to the system are controlled by device drivers. The device drivers contain routines that specify the functions that can be performed by a device, such as read, write, open, and close. Each device has a set of driver routines that can be accessed by the kernel via a device switch table. The kernel uses the major device number designated in a corresponding special file as an index into the device switch table as shown in the next figure. The minor number, which is passed as a parameter, selects one of a class of devices (such as a diskette drive) from a group of devices or specifies device characteristics.

Device Switch Table


Major Number
Routines
devmaj (used to select a set of entry points)
@open @close. . . . .@opencnt
12345 @close @init. . . . .
Address of open routine for selected device

After the kernel creates the i-node table entry for the device, all references to the device use the i-node number assigned to that device until the device is closed.

The following describes an overview of the processing of an I/O request to a device. When a call is made to a device, the kernel first runs the device-independent routines needed for the I/O request. Then, it determines the proper device driver routine to invoke for the required device-dependent process using the major number. Next, it calls the appropriate device driver routine. The requested I/O function is performed, control returns to the kernel. The kernel finishes processing the I/O request and then returns control and any values to the user program.

When a device signals an interrupt to the processor (indicating I/O request completed), control is transferred to the interrupt vector in low memory. The interrupt vector first transfers control to the interrupt handler, which performs device-independent interrupt processing. Next, the device-dependent interrupt handler, which is part of the device driver software, is invoked. The interrupt handler processes the interrupt and then returns control to the kernel. The kernel returns control to the process that had control of the processor at the time of the interrupt.

Chapter 2. System Calls

About This Chapter

This chapter gives detailed information about each of the system calls that are available in the AIX Operating System. System calls provide controlled access to the operating system kernel.

The programming interface to the system calls is identical to that of subroutines. Thus, as far as a C-language program is concerned, a system call is merely a subroutine call. The real difference between a system call and a subroutine is the type of operation it performs. When a program invokes a system call, a context switch takes place so that the called routine has access to the operating system kernel's delicate information. The routine then operates in kernel mode to perform a task on behalf of the program. In this way, access to the delicate system information is restricted to a pre-defined set of routines whose actions can be controlled.

The operations performed by system calls are frequently more basic or "primitive" than those of subroutines. Many subroutines described in Chapter 3, "Subroutines," use system calls to perform more complex tasks. For example, the **open**, **close**, **read**, and **write**, system calls perform very simple I/O operations; but many programs use a standard set of I/O subroutines that add data buffering to the I/O performed by the system calls. (See "standard i/o library" on page 3-342 for details about the Standard I/O Package.)

When an error occurs, most system calls return a value of -1 and set an external variable named **errno** to identify the error. The **errno.h** header file declares the **errno** variable and defines a constant for each of the possible error conditions. A complete listing of these error codes and their meanings can be found in Appendix A, "Error Codes." The specific meanings of the error codes that apply to each system call are listed in the "Diagnostics" section of each system call entry.

For an explanation of the "Syntax" section of each entry, see "Syntax" on page v. For an explanation of header files, see "Header Files" on page vii.

The following discussion is divided into sections that discuss groups of system calls that perform various operations.

Input/Output

The following system calls perform the basic input/output for all types of devices:

access	Determines whether the process has permission to access a file.
chdir	Changes the current directory.
close	Closes a file.
creat	Creates a new file or replaces an existing file with an empty one.
dup	Duplicates an open file descriptor.
fclear	Clears space in a file, freeing unused disk space.
fsync	Forces changes to a file to be written to the disk.
ftruncate	Shortens a file.
ioctl	Provides device-specific control.
lockf	Locks a region of a file from access by other processes.
lseek	Moves the read/write pointer of a file.
open	Opens a file or device for reading or writing.
read	Reads data from a file or device.
write	Writes data to a file or device.

File Maintenance

The file maintenance calls change the access permissions of files, create directories, mount file systems, and perform a variety of other operations:

chmod	Changes the access permission mode of a file.
chown	Changes the user and group that own a file.
chroot	Changes the directory considered to be the root directory.
fcntl	Provides file control.
fstat	Gets file status information.
link	Creates a new directory entry that links to a file.
mknod	Creates a special file that describes a device.
mount	Mounts a file system.
stat	Gets file status information.
sync	Forces all changes in the file system to be written to disk.
umask	Sets the file creation mask.
umount	Unmounts a file system.
unlink	Removes a directory entry.
ustat	Gets file system statistics.
utime	Set the access and modification times of a file.

Process Control

The following system calls control creating, operating and stopping processes:

exec	Replaces the current process image with a new program.
exit	Terminates the current process.
fork	Creates and starts a child process.
nice	Changes the execution priority of a process.
plock	Locks a process in memory.
wait	Waits for a child process to stop or terminate.

Process Identification

The following system calls get and set the IDs and limits of a process:

getegid	Gets the effective group ID.
geteuid	Gets the effective user ID.
getgid	Gets the real group ID.
getgroup	Gets the group access list.
getpgrp	Gets the process group ID.
getpid	Gets the process ID.
getppid	Gets parent process ID.
getuid	Gets the real user ID.
setgid	Sets the real and effective group IDs.
setgroups	Sets the group access list.
setpgrp	Sets the process group ID.
setuid	Sets the real and effective user IDs.
ulimit	Gets and sets the process's user limits.

Signals

Signals are sent to processes when exceptional events occur. A signal interrupts the activity that a process is performing and causes it to take a special action. For example, when a user presses the **Alt-Pause** key sequence at a work station, the **SIGINT** signal is sent to the user's processes. Normally, this causes them to terminate, but each process can arrange to ignore the signal, or to take some other action. The signals that can occur are defined in the **sys/signal.h** header file, and they are further described in "signal" on page 2-145.

Standard signal processing is compatible with UNIX System V and is described in more detail in "signal" on page 2-145. The following system calls handle standard signal processing:

alarm	Sets the process's alarm clock.
kill	Sends a signal to one or more processes.

pause	Suspends the process until a signal arrives.
signal	Sets the action to take when the process receives a signal.

Enhanced signal processing adds several useful features to the facility. It is described in more detail in "Signals" on page 1-20. The following system calls control enhanced signal processing:

sigblock	Blocks signals from being received.
sigsetmask	Sets the signal mask.
sigpause	Suspends the process until a signal arrives.
sigstack	Specifies an alternate stack upon which to process signals.
sigvec	Sets the action to take when the process receives a signal.

Semaphores, Message Queues, and Shared Memory Segments

In addition to signals, the AIX Operating System provides three facilities that provide flexible interprocess communication (IPC): semaphores, message queues, and shared memory segments. Details about the philosophy and use of each these facilities is beyond the scope of this book.

The names of the system calls that deal with semaphores begin with the letters **sem-**. The message queue system calls begin with **msg-**, and the shared memory system calls begin with **shm-**. All three facilities are accessed in a similar manner. The steps are outlined here in approximately the order that they appear in programs:

1. The user specifies a *key* to identify the individual semaphore set, message queue, or shared segment to be accessed. This key is analogous to a file name in that it has been previously agreed upon to identify a specific data structure.

   The key **IPC_PRIVATE** (defined in the **sys/ipc.h** header file) is a special key value that specifies that the data structure is to be private to the current process.

   Keys can be generated by any algorithm as long as the same algorithm is used by all processes on the system. The **ftok** subroutine provides a standard algorithm for generating IPC keys. (See "ftok" on page 3-198 for information about this subroutine.)

2. System calls whose names end with **-get** (**semget**, **msgget**, and **shmget**) use the key to obtain access to the requested data structure. The **-get** system calls are analogous to **open**: each returns an integer identifier (analogous to a file descriptor) that identifies the data structure for access with other system calls.

   Normally, if the semaphore, message queue, or shared segment does not already exist, then the **-get** system call creates the necessary data structure. If another process has already created the data structure by calling the same **-get** system call with the same *key*, then the the identifier of that data structure is returned. This action can be modified with the *semflg*, *msgflg*, or *shmflg* parameter.

However, if **IPC_PRIVATE** is specified as the key, then a private data structure is created. No key exists with which to identify this data structure, so only processes that have its identifier can access it. The current process must pass the identifier to other processes that are to access it. For example, the identifier can be passed to a child process through the *argv* argument vector (see "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34 for details).

3. Shared memory segments must next be attached using the **shmat** system call.
4. The **semop** system call accesses semaphores. Message queues are accessed by **msgsnd**, **msgrcv**, and **msgxrcv**. Programs can access shared memory segments as regular memory through the pointer returned by the **shmat** system call.
5. System calls whose names end with **-ctl** (**semctl**, **msgctl**, and **shmctl**) perform a variety of control operations on the data structure. These control operations include getting status information and changing the access permissions. The data structure associated with each type of IPC identifier is defined in the description of the corresponding **-ctl** system call.
6. When no longer in use, shared memory segments must be detached using the **shmdt** system call.
7. The IPC identifier and the associated data structure should then be removed from the system with the **IPC_RMID** operation of the corresponding **-ctl** system call.

Each IPC data structure contains an **ipc_perm** structure, which contains access permission information. The **ipc_perm** structure is defined in the **sys/ipc.h** header file and it contains the following members:

```
ushort uid;   /* Owner's user ID */
ushort gid;   /* Owner's group ID */
ushort cuid;  /* Creator's user ID */
ushort cgid;  /* Creator's group ID */
ushort mode;  /* Access permission mode */
ushort seq;   /* Slot usage sequence number */
key_t  key;   /* Key */
```

The access permission mechanism resembles the one for files, except that execute permission does not exist for IPC facilities. The **semget**, **msgget**, and **shmget** system calls set the initial permissions when they create new IPC data structures. Also, the user (group) permissions apply if the process's effective user (group) ID matches either **uid (gid)** or **cuid (cgid)**. The permissions can be changed with the corresponding **-ctl** system calls. The **uid** and **gid** fields identify the user and group that own the file for determining whether a given process may access a data structure. The **cuid** and **cgid** fields identify the process that created the data structure, and they can not be changed.

The **mode** field is constructed by logically OR-ing one or more of the following values. Note that these values are defined in the **sys/stat.h** header file and that they are a subset of the access permissions that apply to files.

S_IRUSR Permits the process that owns the data structure to read it.
S_IWUSR Permits the process that owns the data structure to modify it.
S_IRGRP Permits the group associated with the data structure to read it.
S_IWGRP Permits the group associated with the data structure to modify it.
S_IROTH Permits others to read the data structure.
S_IWOTH Permits others to modify the data structure.

For more information about the interprocess communication facilities, see *AIX Operating System Programming Tools and Interfaces.*

The **shmat** system call can be used to attach mapped files as well as shared memory segments. The two functions are similar in some ways, but they should not be confused. For an overview of the mapped file facility, see page 1-13. For more detailed information, see the following discussion and "shmat" on page 2-131.

Mapped Files

The AIX Operating System allows programs to map a file onto a memory segment so that that data in the file can be accessed more quickly and more directly. See page 1-13 for an overview of the file-mapping facility.

The steps for setting up and accessing a mapped file are outlined here in the order that they occur in programs:

1. Open the file with the **open** system call. The file must be a regular file. Directories and special files cannot be mapped.

2. Attach the file to a memory segment using the **shmat** system call. Specify the file descriptor returned by **open** in place of the shared memory identifier as the *shmid* parameter. The *shmflg* parameter is either **SHM_MAP** (to select file mapping), or it is constructed by logically OR-ing the value **SHM_MAP** with one of the following values:

   SHM_RDONLY Maps the file in read-only mode.
   SHM_COPY Maps the file in copy-on-write mode.

   If neither **SHM_RDONLY** nor **SHM_COPY** is set, then the file is mapped in read-write mode. The *shmflg* parameter can also be logically OR-ed with the following value:

   SHM_RND Rounds the address specified by the *shmaddr* parameter to the next lowest segment boundary, if necessary.

   The file must be opened for writing (in step 1) before it can be mapped read-write or copy-on-write.

3. Access shared memory segments as regular memory through the pointer returned by the **shmat** system call. The system performs the necessary read operations for you automatically. If the file is mapped read-write, then the system automatically writes to the file as well.

4. If the file is mapped copy-on-write, then you must explicitly tell the system to update the file by using the **fsync** system call. If you never call **fsync**, then changes made to the mapped file in memory are never written to permanent storage.
5. The **shmctl** system call can be used to get status information about the memory segment onto which the file is mapped.
6. If you wish, you can use the **shmdt** system call to unmap the file and detatch the memory segment, leaving the file open for conventional I/O.
7. Close the file with the **close** system call. **close** automatically detatches the memory segment (unless you already did this in step 6).

Mapped files can be shared with other processes that map the file, or that use the conventional I/O system calls. All of these processes access the same shared memory segment, except for those that write to the file after mapping it in copy-on-write mode. Each process that maps a file copy-on-write gets a private mapped copy of the file when it first attempt to write to it.

Warning: Data may be lost if a process modifies a file that another process has mapped copy-on-write. When the latter process calls **fsync**, the changes made by the former process are overwritten.

access

Purpose

Determines the accessibility of a file.

Syntax

#include <unistd.h>

int access (*path***,** *amode***)**
char ****path*;**
int *amode***;**

Description

The **access** system call checks the accessibility of the file specified by the *path* parameter, using the real user ID in place of the effective user ID and the real group ID in place of the effective group ID. If Distributed Services is installed on your system, this path can cross into another node. The **access** system call checks the named file to see if the type of access specified by the *amode* parameter is permitted.

The bit pattern contained in *amode* is constructed by logically OR-ing the following values:

R_OK	Checks read permission.
W_OK	Checks write permission.
X_OK	Checks execute (search) permission.
F_OK	Checks to see if the file exists.

The owner of a file has access checked with respect to the owner read, write, and execute mode bits. Members of the file's group other than the owner have access checked with respect to the group mode bits. All others have access checked with respect to the other mode bits.

Return Value

If the requested access is permitted, a value of 0 is returned. If the requested access is denied, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

Access to the file is denied if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT Read, write, or execute (search) permission is requested for a null path name.

ENOENT The named file does not exist.

EACCES Search permission is denied on a component of the path prefix.

EACCES Permission bits of the file mode do not permit the requested access.

EROFS Write access is requested for a file on a read-only file system.

ETXTBSY Write access is requested for a pure procedure (shared text) file that is being executed.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **access** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: “chmod” on page 2-18 and “stat, fstat” on page 2-159.

acct

Purpose

Enables and disables process accounting.

Syntax

int acct (*path*)
char *_path_;

Description

The **acct** system call enables the accounting routine when the *path* parameter specifies the path name of the file to which an accounting record is written for each process that terminates. (For information about the accounting file, see "acct" on page 4-15.) When the *path* parameter is 0 or **NULL**, the **acct** system call disables the accounting routine.

If Distributed Services is installed on your system, the accounting file can reside on another node.

Warning: To insure accurate accounting, each node must have its own accounting file, which can be located on any node in the network.

The effective user ID of the calling process must be superuser to use the **acct** system call.

Return Value

Upon successful completion, **acct** returns a value of 0. If **acct** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **acct** system call fails if one or more of the following are true:

EPERM	The effective user ID of the calling process is not superuser.
EBUSY	An attempt is made to enable accounting when it is already enabled.
ENOTDIR	A component of the *path* parameter is not a directory.
ENOENT	Any component of the accounting file's path name does not exist.

EACCES Any component of the *path* parameter denies search permission.

EACCES The file named by the *path* parameter is not an ordinary file.

EACCES Mode permission is denied for the named accounting file.

EISDIR The named file is a directory.

EROFS The named file resides on a read-only file system.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **acct** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "exit, _exit" on page 2-40, "signal" on page 2-145, and "acct" on page 4-15.

The discussion of **acct** in *Managing the AIX Operating System.*

alarm

Purpose

Sets a process's alarm clock.

Syntax

unsigned int alarm (*sec***)**
unsigned int *sec***;**

Description

The **alarm** system call instructs the calling process's alarm clock to send a **SIGALRM** signal to the calling process after the number of real-time seconds specified by the *sec* parameter have elapsed. (See "signal" on page 2-145 for more information about signals.)

The **alarm** system calls are not stacked. Successive **alarm** system calls reset the calling process's alarm clock.

If the *sec* parameter is 0, any previous alarm request is canceled.

Return Value

The **alarm** system call returns the amount of time previously remaining in the calling process's alarm clock. If no **alarm** request was previously issued, then a value of 0 is returned.

Related Information

In this book: "pause" on page 2-94 and "signal" on page 2-145.

brk, sbrk

Purpose

Changes data segment space allocation.

Syntax

int brk (*endds***)** **char ****endds***;**	**char *sbrk (***incr***)** **int** *incr***;**

Description

The **brk** and **sbrk** system calls dynamically change the amount of space allocated for the calling process's data segment. (For information about data segments, see "exec: execl, execv, execle, execvc, execlp, execvp" on page 2-34.)

The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the current end of the data segment. The amount of allocated space increases as the break value increases. The newly allocated space is initialized to 0. The break value can be automatically rounded up to a size appropriate for the memory management architecture.

The **brk** system call sets the break value to the value of the *endds* parameter and changes the allocated space accordingly.

The **sbrk** system call adds to the break value the number of bytes contained in the *incr* parameter and changes the allocated space accordingly. The *incr* parameter can be a negative number, in which case the amount of allocated space is decreased.

Return Value

Upon successful completion, the **brk** system call returns a value of 0, and the **sbrk** system call returns the old break value. If the **brk** or the **sbrk** system calls fail, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **brk** and the **sbrk** system calls fail and the allocated space remains unchanged if one or more of the following are true:

ENOMEM The requested change will allocate more space than is allowed by a system-imposed maximum. (For information on the system-imposed maximum on memory space, see "ulimit" on page 2-167.)

ENOMEM The requested change will set the break value to a value greater than or equal to the start address of any attached shared memory segment. (For information on shared memory operations, see "shmat" on page 2-131, "shmdt" on page 2-138, and "shmget" on page 2-140.)

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "shmat" on page 2-131, "shmdt" on page 2-138, and "ulimit" on page 2-167.

chdir

Purpose

Changes the current directory.

Syntax

int chdir (*path*)
char **path*;

Description

The **chdir** system call changes the current directory to the directory specified by the *path* parameter. If Distributed Services is installed on your system, this path can cross into another node. The ***current directory***, also called the ***current working directory***, is the starting point of searches for path names that do not begin with a / (slash).

Return Value

Upon successful completion, the **chdir** system call returns a value of 0. If the **chdir** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **chdir** system call fails and the current directory remains unchanged if one or more of the following are true:

ENOTDIR	A component of the *path* parameter is not a directory.
ENOENT	The named directory does not exist.
EACCES	Search permission is denied for any component of the *path* parameter.
EFAULT	The *path* parameter points to a location outside of the process's allocated address space.
ESTALE	The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **chdir** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EDIST	The server has a release level of Distributed Services that cannot communicate with this node.
EAGAIN	The server is too busy to accept the request.
ESTALE	The file descriptor for a remote file has become obsolete.
EPERM	The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.
ENODEV	The named file is a remote file located on a device that has been unmounted at the server.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
ENOCONNECT	An attempt to establish a new network connection with a remote node failed.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chroot" on page 2-23.

The **cd** command in *AIX Operating System Commands Reference.*

chmod

Purpose

Changes file access permissions.

Syntax

#include <sys/stat.h>

int chmod (*path***,** *mode***)**
char ****path*;**
int *mode***;**

Description

The **chmod** system call sets the access permissions of the file specified by the *path* parameter. If Distributed Services is installed on your system, this path can cross into another node. The access permissions of the file are set according to the bit pattern specified by the *mode* parameter.

To change file access permissions, the effective user ID of the calling process must either be superuser or match the ID of the file's owner.

The *mode* parameter is constructed by logically OR-ing one or more of the following values, which are defined in the **sys/stat.h** header file:

S_ISUID Sets the process's effective user ID to the file's owner on execution.
S_ISGID Sets the process's effective group ID to the file's group on execution.
S_ISVTX Saves text image after execution.
S_ENFMT Enables enforcement-mode record locking.
S_IRUSR Permits the file's owner to read it.
S_IWUSR Permits the file's owner to write to it.
S_IXUSR Permits the file's owner to execute it (or to search the directory).
S_IRGRP Permits the file's group to read it.
S_IWGRP Permits the file's group to write to it.
S_IXGRP Permits the file's group to execute it (or to search the directory).
S_IROTH Permits others to read the file.
S_IWOTH Permits others to write to the file.
S_IXOTH Permits others to execute the file (or to search the directory).

Other mode values exist that can be set with the **mknod** system call, but not with **chmod**. A complete list of the possible file mode values and other useful macros appears in "stat.h" on page 5-69.

Setting **S_ISVTX** for a shared executable file prevents the system from unmapping the program text segment of the file when its last user terminates. Thus, when the next process executes it, the text need not be read from the file system. It is simply paged in, saving time.

If **S_ENFMT** is set and no execute permissions are set, then locks placed on the file with the **lockf** system call are ***enforced locks***. See "lockf" on page 2-64 for details about locking regions of a file.

If the effective user ID of the calling process is not superuser and the file is not a character special file, then the **chmod** system call clears the **S_ISVTX** bit.

If the effective user ID of the process is not that of superuser, and if the effective group ID or one of the IDs in the group access list of the process does not match the file's existing group ID, then the **chmod** system call clears the **S_ISGID** bit. (See "getgroups" on page 2-52 and "setgroups" on page 2-126 for more information about the group access list.)

Return Value

Upon successful completion, the **chmod** system call returns a value of 0. If the **chmod** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **chmod** system call fails and the file permissions remain unchanged if one or more of the following are true:

ENOTDIR A component of the *path* parameter is not a directory.

ENOENT The named file does not exist.

EACCES A component of the *path* parameter has search permission denied.

EPERM The effective user ID does not match the ID of the owner of the file or the ID of superuser.

EROFS The named file resides on a read-only file system.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **chmod** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EDIST	The server has a release level of Distributed Services that cannot communicate with this node.
EAGAIN	The server is too busy to accept the request.
ESTALE	The file descriptor for a remote file has become obsolete.
EPERM	The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.
ENODEV	The named file is a remote file located on a device that has been unmounted at the server.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
ENOCONNECT	An attempt to establish a new network connection with a remote node failed.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chown, chownx" on page 2-21, "getgroups" on page 2-52, "mknod" on page 2-69, and "setgroups" on page 2-126.

The **chmod** command in *AIX Operating System Commands Reference.*

chown, chownx

Purpose

Changes the owner and group IDs of a file.

Syntax

int chown (*path*, *owner*, *group***)**
**char ****path*;
int *owner*, *group*;

#include <sys/chownx.h>

int chownx (*path*, *owner*, *group*, *tflag***)**
**char ****path*;
int *owner*, *group*, *tflag*;

Description

The **chown** system call changes the owner ID and the group ID of the file named by the *path* parameter. If Distributed Services is installed on your system, the path can cross into another node, naming a remote file.

If the named file is a local file, the owner and group IDs of that file are set to the numeric values contained in the *owner* and *group* parameters, respectively. If the named file is a remote file, then the IDs of the named file are set to the values contained in owner and group *after* both outbound and inbound translation. (See *Managing the AIX Operating System* for a description of ID translation.)

A process can change the ownership of a file only if its effective user ID (translated, if the file is remote) is either superuser or the same as the file's owner ID.

If the effective user ID of the calling process is not the same as the superuser ID, then the **chown** system call clears the set-user-ID (**S_ISUID**) and set-group-ID (**S_ISGID**) bits of the file mode. (See "stat.h" on page 5-69 for the definitions of the constants **S_ISUID** and **S_ISGID**.)

The **chownx** system call performs the same function as the **chown** system call, except that it also allows the process to change owner and group IDs with or without ID translation by specifying the *tflag* parameter.

The *tflag* parameter determines the translation of the *owner* and *group* parameters. This parameter is constructed by logically ORing two of the following values:

T_OWNER_RAW Changes the file's owner to the value of the *owner* parameter without translation.

T_OWNER_TRAN Changes the file's owner to the value of the *owner* parameter after translation through the sending node's outbound translate tables and the receiving node's inbound translation tables. If the file is a local file, this is the same as **T_OWNER_RAW**; no translation is done.

T_OWNER_AS_IS Ignores the value specified in the *owner* parameter and leaves the owner ID of the file unaltered.

T_GROUP_RAW Changes the file's group ID to the value of the *group* parameter without translation.

T_GROUP_TRAN Changes the file's group ID to the value of the *group* parameter after translation through the sending node's outbound translate tables and the receiving node's inbound translation tables. If the file is a local file, this is the same as **T_GROUP_RAW**; no translation is done.

T_GROUP_AS_IS Ignores the value specified in the *group* parameter and leaves the group ID of the file unaltered.

Only one each of the **T_OWNER** and the **T_GROUP** bits should be specified.

Note that the following two system calls are equivalent:

chown (*path*, *owner*, *group*)

chownx (*path*, *owner*, *group*, **T_OWNER_TRAN** | **T_GROUP_TRAN**)

Return Value

Upon successful completion, a value of 0 is returned. If the **chown** or **chownx** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **chown** and **chownx** system calls fail and the owner ID and the group ID of the named file remain unchanged if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

EACCES Search permission is denied on a component of the path prefix.

EPERM The effective user ID does not match the owner of the file and the effective user ID is not superuser.

EROFS The named file resides on a read-only file system.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **chown** or **chownx** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The server's translate tables do not contain an entry for at least one of the following IDs:

- The user ID of the caller
- The *owner* parameter
- The *group* parameter.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "fullstat, ffullstat" on page 2-50.2, and "stat.h" on page 5-69.

The **chown** command in *AIX Operating System Commands Reference.*

Managing the AIX Operating System.

chroot

Purpose

Changes the effective root directory.

Syntax

int chroot (*path*)
char ****path*;**

Description

The **chroot** system call causes the directory named by the *path* parameter to become the effective root directory. If Distributed Services is installed on your system, this path can cross into another node. The ***effective root directory*** is the starting point when searching for a file whose path name begins with / (slash). The current directory is not affected by the **chroot** system call.

The effective user ID of the calling process must be superuser to change the effective root directory.

The . . (dot-dot) entry in the effective root directory is interpreted to mean the effective root directory itself. Thus, . . (dot-dot) cannot be used to access files outside the subtree rooted at the effective root directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **chroot** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **chroot** system call fails and the effective root directory remains unchanged if one or more of the following are true:

ENOTDIR	Any component of the path name is not a directory.
ENOENT	The named directory does not exist.
EPERM	The effective user ID of the calling process is not superuser.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **chroot** can also fail if one or more of the following are true:

EACCES Search permission was denied on a component of the path.

The effective user ID of the calling process must be the same as the superuser ID to issue this call. Since with Distributed Services *path* can cross into another node and a process that has superuser authority in the local node probably does not have superuser authority in the remote node, search permission may be denied even to the local superuser.

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: “chdir” on page 2-16.

The **chroot** command in *AIX Operating System Commands Reference.*

close

Purpose

Closes the file associated with a file descriptor.

Syntax

int close (*fildes*)
int *fildes*;

Description

The **close** system call closes the file associated with the file descriptor *fildes*. If Distributed Services is installed on your system, this file can reside on another node. The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call.

All file regions that this process has previously locked with the **lockf** system call are unlocked. This includes regions of files other than the file specified by the *fildes* parameter.

If the *fildes* parameter is associated with a mapped file, and if no other process has attached this mapped file, then it is unmapped.

Return Value

Upon successful completion, a value of 0 is returned. If the **close** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **close** system call fails if the following is true:

EBADF The *fildes* parameter is not a valid open file descriptor.

If Distributed Services is installed on your system, **close** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed. Data may be lost.

This **errno** value occurs only when the connection has been lost *and* there is data stored at the client that cannot be written to the server. Otherwise, having a lost connection at close time does not cause an error. (For example, in cases where a read-only file is closed.)

Related Information

In this book: "creat" on page 2-27, "dup" on page 2-32, "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fcntl" on page 2-44, "open" on page 2-90, "pipe" on page 2-95, and Appendix C, "Writing Device Drivers."

creat

Purpose

Creates a new file or rewrites an existing file.

Syntax

#include <stat.h>

int creat (*path*, *mode*)
char *_path_;
int _mode_;

Description

The **creat** system call creates a new ordinary file or prepares to rewrite an existing file named by the *path* parameter. If Distributed Services is installed on your system, this path can cross into another node.

If the file exists, the length is truncated to 0 and the mode and owner are unchanged. If the file does not exist, the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of the *mode* parameter modified as follows:

- All bits set in the process's file mode creation mask are cleared. (For information about the file mode creation mask, see "umask" on page 2-169.)
- The ***save-text-image-after-execution bit*** of the file mode (**S_ISVTX**) is cleared. (For more information about this bit, see "chmod" on page 2-18.)

See "chmod" on page 2-18 for a detailed explanation of file modes.

No process can have more than 200 files open simultaneously. A new file can be created with a mode that forbids writing.

Note that the following two system calls are equivalent:

> **creat (*path*, *mode*)**
>
> **open (*path*, O_WRONLY | O_CREAT | O_TRUNC, *mode*)**

See "open" on page 2-90 for details about the **open** system call.

Return Value

Upon successful completion, a file descriptor (a nonnegative integer) is returned and the file is opened for writing, even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across **exec** system calls. (For information about control of open files, see “fcntl” on page 2-44.)

If the **creat** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **creat** system call fails if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist.

EACCES Search permission is denied on a component of the path prefix.

ENOENT The path name is null.

EACCES The file does not exist and the directory in which the file is to be created does not permit writing.

EROFS The named file resides or would reside on a read-only file system.

ETXTBSY The file is a pure procedure (shared text) file that is being executed.

EACCES The file exists and write permission is denied.

EISDIR The named file is an existing directory.

EMFILE Two hundred (200) file descriptors are currently open.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ENFILE The system file table is full.

EAGAIN The named file contains a record lock owned by another process. See “lockf” on page 2-64 for information about record locks.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **creat** can also fail if one or more of the following are true:

EINVAL	The *path* parameter identifies a remote file that is neither a directory nor a regular file.
EBUSY	The special file to open for writing is already mounted.
EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EDIST	The server has a release level of Distributed Services that cannot communicate with this node.
EAGAIN	The server is too busy to accept the request.
ESTALE	The file descriptor for a remote file has become obsolete.
EPERM	The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.
ENODEV	The named file is a remote file located on a device that has been unmounted at the server.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
ENOCONNECT	An attempt to establish a new network connection with a remote node failed.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "close" on page 2-25, "dup" on page 2-32, "lseek" on page 2-67, "open" on page 2-90, "read, readx" on page 2-106, "umask" on page 2-169, and "write, writex" on page 2-184.

disclaim

Purpose

Disclaims content of a memory address range.

Syntax

#include <sys/shm.h>

int disclaim (*addr***,** *length***,** *flag***)**
char ****addr*;**
unsigned int *length***,** *flag***;**

Description

The **disclaim** system call marks an area of memory that has content that is no longer needed. This allows the system to discontinue paging the memory area. This system call cannot be used on memory that is mapped to a file by the **shmat** system call.

The *addr* parameter points to the beginning of the memory area, and the *length* parameter specifies its length in bytes. The *flag* parameter must be the value **ZERO_MEM**, which indicates that each memory location in the address range is to be set to 0.

Return Value

Upon successful completion, the **disclaim** system call returns a value of 0. If it fails, it returns a value of -1 and sets **errno** to indicate the error.

Diagnostics

The **disclaim** system call fails if one or more of the following is true:

EFAULT The calling process does not have write access to the area of memory that begins at *address* and extends for *length* bytes.

EINVAL The value of the *flag* parameter is not valid.

EINVAL The memory area is mapped to a file.

Related Information

In this book: "shmat" on page 2-131 and "shmctl" on page 2-135.

dsstate

Purpose

Controls the kernel operations related to Distributed Services.

Syntax

#include <sys/dsstate.h>

int dsstate (*buf*)
struct dsstate **buf*;

Description

The **dsstate** system call controls the kernel operations related to Distributed Services. A process with an effective user ID of superuser can use the **dsstate** system call to change the state of the kernel, while a process that does not have superuser privileges can use it to query the state of the kernel. The *buf* parameter is a pointer to a structure of type **ds_state**. This structure is defined in the **sys/dsstate.h** header file, and it contains the following members:

```
short i_state;     /* input state                 */
short i_kprocs;    /* input number of kprocs      */
short r_state;     /* result state                */
short r_kprocs;    /* result number of kprocs     */
int   reserved[4]   /* reserved                    */
                   /* each element must be zero   */
```

The following bits are valid in the **i_state** bit field:

DS_START_SERV_SYNC Causes updates to all files for which this node is a server to be written directly to the server, rather than to storage at the client node. All writes to files are sent to this file server, and all reads from files are provided by the server.

DS_END_SERV_SYNC Allows clients of this node to store data instead of forcing all storage to take place on the server node.

DS_START_CLIENT_SYNC Causes updates to all files for which this node is a client to be written directly to the server, rather than to local storage. All writes to files are sent to the file server, and all reads from files are provided by the server. When

CLIENT_SYNC is first started, all data stored at the client (local node) is written to the server where each remote file resides, and **dsstate** does not return until these writes are finished.

DS_END_CLIENT_SYNC Allows some data to be stored at the local node, rather than processing all reads and writes through the server.

DS_BLOCK_SERV Causes this server to reject all requests for file services, including both new requests and requests for files already in use. When **BLOCK_SERV** is entered, this server forces any data stored on client nodes to be written to the server before any server requests are rejected.

DS_ALLOW_SERV Allows this server to accept requests for file services from other nodes.

DS_BLOCK_ALL Causes all data from this client node to be written to the appropriate server, then breaks all existing connections with remote nodes and rejects new remote requests. **DS_BLOCK_ALL** sets the number of kernel processes to 0.

DS_ALLOW_ALL Allows remote requests. Connections with remote nodes can then be established as needed.

DS_STARTK Starts the kernel processes for Distributed Services. **DS_STARTK** must be set the first time the **dsstate** system call is used, and other fields can either be set at the same time or with later calls.

The **i_kprocs** field sets the number of active Distributed Services kernel processes. If the value of **i_kprocs** exceeds the number of kernel processes allocated to Distributed Services when the system was initialized, then only the available processes are started. If **i_kprocs** is 0 or negative, then the number of active kernel processes is not changed. (To alter the state bits without changing the number of kernel processes, set **i_kprocs** to 0.) If **i_kprocs** is greater than 0, either **DS_STARTK** must be set with this **dsstate** system call or must have been set by an earlier **dsstate** call.

The bits of the **r_state** field are set to indicate the state of the kernel after the **dsstate** system call has taken effect. The following bits are returned:

ALLOW_ALL When set, this node can make file service requests of other nodes.

ALLOW_SERV When set, this node can accept requests for file services from other nodes.

CLIENT_SYNC When set, all files for which this node is a client are written directly to the server, rather than stored locally.

SERVER_SYNC When set, all data for which this node is a server is written directly to the server, rather than stored at the client node.

STARTK When set, the Distributed Services kernel processes, if any were specified in **i_kprocs**, are started.

DSINITED When set, the Distributed Services kernel processes have been started. The setting of this bit is always the same as the **STARTK** bit.

If **i_state** does not have either bit set for a given pair of values (such as **DS_BLOCK_ALL** and **DS_ALLOW_ALL**), then the current state of that pair is not altered and is returned in **r_state**. Otherwise, **r_state** contains the value that was set on the last call to **dsstate**.

The default state for the **BLOCK/ALLOW** pairs is to **ALLOW** requests, while the default for the **START_SYNC/END_SYNC** pairs is **END_SYNC**.

The **r_kprocs** field is set to indicate the number of kernel processes after the **dsstate** system call has taken effect.

If the effective user ID of the calling process is not superuser, then **dsstate** ignores **i_state** and **i_kprocs**, and the current state remains unaltered. A process without superuser privileges can, however, determine the current kernel state by examining the contents of **r_state** and **r_kprocs** at the end of a **dsstate** system call.

Return Value

Upon successful completion, the **dsstate** system call returns a value of 0. If the **dsstate** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **dsstate** system call fails and the state and number of kernel processes remain unchanged if the following is true:

EINVAL Invalid input data (such as mutually exclusive parameters).

ENOMEM Not enough kernel processes are available to run Distributed Services.

Related Information

In this book: "master" on page 4-98.

Managing the AIX Operating System.

dup

Purpose

Duplicates an open file descriptor.

Syntax

int dup (*fildes*)
int *fildes*;

Description

The **dup** system call returns a new file descriptor for the file descriptor pointed to by the *fildes* parameter. The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The **dup** system call returns a new file descriptor having the following in common with the original:

- The same open file or pipe
- The same file pointer (that is, both file descriptors share one file pointer)
- The same access mode (read, write or read/write)
- The same file status flags
- The same locks.

The new file descriptor is set to remain open across **exec** system calls. (For more information about file control, see "fcntl.h" on page 5-56.)

The file descriptor returned is the lowest one available.

Return Value

Upon successful completion, a file descriptor (nonnegative integer) is returned. If the **dup** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **dup** system call fails if one or more of the following are true:

EBADF	*fildes* is not a valid open file descriptor.
EMFILE	Two hundred (200) file descriptors are currently open.

Related Information

In this book: "close" on page 2-25, "creat" on page 2-27, "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "open" on page 2-90, "pipe" on page 2-95, and "fcntl.h" on page 5-56.

exec: execl, execv, execle, execve, execlp, execvp

Purpose

Executes a file.

Syntax

int execl (*path***,** *arg0* [, *arg1*, . . .], **0)**
char **path*, **arg0*, **arg1*, . . . ;

int execle (*path***,** *arg0* [, *arg1*, . . .], **0,** *envp***)**
char **path*, **arg0*, **arg1*, . . . , **envp*[ ];

int execlp (*file***,** *arg0* [, *arg1*, . . .], **0)**
char **file*, **arg0*, **arg1*, . . . ;

int execv (*path***,** *argv***)**
char **path*, **argv*[ ];

int execve (*path***,** *argv***,** *envp***)**
char **path*, **argv*[ ], **envp*[ ];

int execvp (*file***,** *argv***)**
char **file*, **argv*[ ];

Description

The **exec** system call, in all its forms, executes a new program in the calling process. **exec** does not create a new program, but overlays the current program with a new one, which is called the ***new process image***. The new process image file can be one of three file types:

- An executable binary file in **a.out** format (see “a.out” on page 4-5)
- An executable text file that contains a shell procedure (only **execlp** and **execvp** allow this type of new process image file)
- A file that names an executable binary file or shell procedure to be run.

The last of the types mentioned is recognized by a header with the syntax:

#! *path* [*string*]

The #! is the file's ***magic number***, which identifies the file type. *path* is the path name of the file to be executed. If Distributed Services is installed on your system, this path can cross into another node. *string* is an optional character string that contains no tab or space characters. If specified, this string is passed to the new process as an argument in front of the name of the new process image file. The header must be terminated with a new-line character. When invoked, the new process is passed *path* as *argv*[0]. If a *string* is specified in the new process image file, then the **exec** system call sets *argv*[0] to *string* and *path* concatenated together. The rest of the arguments passed are the same as those passed to the **exec** system call.

The parameters for the **exec** system calls are defined as follows:

path This parameter points to the path name of the new process image file. If Distributed Services is installed on your system, this path can cross into another node. Data is copied into local virtual memory before proceeding.

file This parameter points to the name of the new process image file. Unless *file* is a full path name, the path prefix for the file is obtained by searching the directories named in the **PATH** environment variable. The initial environment is supplied by the shell.

Note that **execlp** and **execvp** take *file* parameters, but the rest of the **exec** system calls take *path* parameters. (For information about the environment, see "environment" on page 5-47 and the **sh** command in *AIX Operating System Commands Reference.*)

arg0 [, *arg1, . . .*]
These parameters point to null-terminated character strings. The strings constitute the argument list available to the new process. By convention, at least *arg0* must be present, and it must point to a string that is the same as *path* or its last component.

argv This parameter is an array of pointers to null-terminated character strings. These strings constitute the argument list available to the new process. By convention, *argv* must have at least one element, and it must point to a string that is the same as *path* or its last component. The last element of *argv* is a **NULL** pointer.

envp This parameter is an array of pointers to null-terminated character strings. These strings constitute the ***environment*** for the new process. The last element of *envp* is a **NULL** pointer.

When a C program is executed, it receives the following parameters:

```
main (argc, argv, envp)
int argc;
char *argv[ ], *envp[ ];
```

Here *argc* is the argument count, and *argv* is an array of character pointers to the arguments themselves. By convention, the value of *argc* is at least one, and *argv*[0] points to a string containing the name of the new process image file.

The **main** routine of a C language program automatically begins with a run-time start-off routine. This routine sets a global variable named **environ** so that it points to the environment array passed to the program in **envp**. You can access this global variable by including the following declaration in your program:

```
extern char **environ;
```

The **execl**, **execv**, **execlp**, and **execvp** system calls use **environ** to pass the calling process's current environment to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose **close-on-exec** flag is set. For those file descriptors that remain open, the file pointer is unchanged. (For information about file control, see "fcntl" on page 2-44.)

If the new process requires shared libraries, **exec** will find, open, and map each shared library image to the new process address space. (See *AIX Operating System Programming Tools and Interfaces.*) Shared libraries are searched for in the directories listed in the **LIBPATH** environment variable. If any of these files is remote, the data is copied into local virtual memory.

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so after **exec**. Ignored signals remain ignored, the signal mask remains the same, and the signal stack state is reset. (For information about signals, see "signal" on page 2-145 and "sigvec" on page 2-156.)

If the ***set-user-ID mode bit*** of the new process image file is set, then **exec** sets the effective user ID of the new process to the owner ID of the new process image file. Similarly, if the ***set-group-ID mode bit*** of the new process image file is set, then the effective group ID of the new process is set to the group ID of the new process image file. The real user ID and real group ID of the new process remain the same as those of the calling process. (For information about the set-ID modes, see "chmod" on page 2-18.)

When one or both of the set-ID mode bits is set and the file to be executed is a remote file, the file's user and group IDs go through outbound translation at the server. Then they are transmitted to the client node where they are translated according to the inbound translation table. These translated IDs become the user and group IDs of the new process. See *Managing the AIX Operating System* for a discussion of UID and GID translation.

The shared libraries attached to the calling process are not attached to the new process. (For information about shared memory segments, see "shmat" on page 2-131, "shmdt" on page 2-138, and "shmget" on page 2-140.)

Profiling is disabled for the new process. (For information about profiling, see "profil" on page 2-99.)

The new process inherits the following attributes from the calling process:

- Nice value (see "nice" on page 2-88)
- Process ID
- Parent process ID
- Process group ID
- semadj values (see "semop" on page 2-122)
- TTY group ID (see "exit, _exit" on page 2-40 and "signal" on page 2-145)
- Trace flag (see request 0 of "ptrace" on page 2-102)
- Time left until an alarm clock signal (see "alarm" on page 2-13)
- Current directory
- Root directory
- File mode creation mask (see "umask" on page 2-169)
- File size limit (see "ulimit" on page 2-167)
- **utime**, **stime**, **cutime**, and **cstime** (see "times" on page 2-165).

Return Value

Upon successful completion, **exec** does not return because the calling process image is overlaid by the new process image. If **exec** returns to the calling process, then it returns the value -1 and sets **errno** to indicate the error.

Diagnostics

The **exec** system call fails and returns to the calling process if one or more of the following are true:

ENOENT One or more components of the new process image file's path name do not exist.

ENOTDIR A component of the path prefix of the new process image file is not a directory.

EACCES Search permission is denied for a directory listed in the path prefix of the new process image file.

EACCES The new process image file is not an ordinary file.

EACCES The mode of the new process image file denies execution permission.

ENOEXEC The exec is not an **execlp** or **execvp**, and the new process image file has the appropriate access permission but has an invalid magic number in its header.

EINVAL The new process image file has a valid magic number in its header, but the header is damaged or is incorrect for the machine on which the file is to be run.

ETXTBSY The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

ENOMEM The new process requires more memory than is allowed by the system-imposed maximum **MAXMEM**.

E2BIG The number of bytes in the new process's argument list is greater than the system-imposed limit. This limit is defined as **NCARGS** in the **sys/param.h** header file.

EFAULT The *path*, *argv*, or *envp* parameter points to a location outside of the process's allocated address space.

In addition, some errors can occur when using the new process file after the old process image has been overwritten. These errors include problems in setting up new data and stack registers, problems in mapping a shared library, or problems in reading the new process file. Because returning to the calling process is not possible, the system sends the **SIGKILL** signal to the process when one of these errors occurs.

If an error occurred while mapping a shared library, an error message describing the reason for failure will be written to standard error before the signal **SIGKILL** is sent to the process. (See *AIX Operating System Programming Tools and Interfaces.*) If a shared library cannot be mapped, one or more of the following is true:

ENOENT One or more components of the path name of the shared library file do not exist.

ENOTDIR A component of the path prefix of the shared library file is not a directory.

EACCES Search permission is denied for a directory listed in the path prefix of the shared library file.

EACCES The shared library file mode denies execution permission.

ENOEXEC The shared library file has the appropriate access permission but an invalid magic number in its header.

ETXTBSY The shared library file is currently open for writing by some other process.

ENOMEM The shared library requires more memory than is allowed by the system-imposed maximum.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **exec** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The set-user-ID or set-group-ID bit is set on the process image file, and the translation tables at the server or client do not allow translation of this user or group ID.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Examples

1. To run a command and pass it a parameter:

   ```
   execlp("li", "li", "-al", 0);
   ```

   The **execlp** system call searches each of the directories listed in the **PATH** environment variable for the **li** command, and then it overlays the current process image with this command. **execlp** does not return, unless the **li** command cannot be executed. Note that this example does not run the shell command processor, so operations interpreted by the shell, such as using wildcard characters in file names, are not valid.

2. To run the shell to interpret a command:

   ```
   execl("/bin/sh", "sh", "-c", "li -l *.c", 0);
   ```

   This runs the **sh** (shell) command with the **-c** parameter, which indicates that the following parameter is the command to be interpreted. (See the discussion of **sh** in *AIX Operating System Commands Reference* for details about this command.) This example uses **execl** instead of **execlp** because the full path name **/bin/sh** is specified, making a **PATH** search unnecessary.

   Running a shell command in a child process is generally more useful than simply using **exec**, as shown here. The simplest way to do this is to use the **system** subroutine. See "system" on page 3-350 for information about this subroutine.

3. The following is an example of a new process file that names a program to be run:

   ```
   #! /usr/bin/awk -f
   { for (i = NF; i > 0; --i)  print i }
   ```

   If this file is named `reverse`, then typing the following command on the command line:

   ```
   reverse chapter1 chapter2
   ```

   causes the following command to be run:

   ```
   /usr/bin/awk -f reverse chapter1 chapter2
   ```

   Note that the **exec** system calls use only the first line of the new process image file and ignore the rest of it. Also, **awk** interprets the text that follows a # (number sign) as a comment. (See the **awk** command in *AIX Operating System Commands Reference* for more information.)

Related Information

In this book: “alarm” on page 2-13, “chmod” on page 2-18, “exit, _exit” on page 2-40, “fcntl” on page 2-44, “fork” on page 2-46, “nice” on page 2-88, “profil” on page 2-99, “ptrace” on page 2-102, “semop” on page 2-122, “shmat” on page 2-131, “signal” on page 2-145, “sigvec” on page 2-156, “times” on page 2-165, “ulimit” on page 2-167, “umask” on page 2-169, “system” on page 3-350, “varargs” on page 3-371, “a.out” on page 4-5, and “environment” on page 5-47.

The **sh** and **shlib** commands in *AIX Operating System Commands Reference.*

exit, _exit

Purpose

Terminates a process.

Syntax

void exit (*status***)**
int *status*;

void _exit (*status***)**
int *status*;

Description

The **exit** system call terminates the calling process and causes the following to occur:

- All of the file descriptors open in the calling process are closed. If Distributed Services is installed on your system, some of these files may be remote. Since **exit** terminates the process, any errors encountered during these close operations go unreported.
- If the parent process of the calling process is executing a **wait** system call, it is notified of the termination of the calling process and the low-order eight bits (that is, bits 0377 or 0xFF) of *status* are made available to it. See "wait" on page 2-182.
- If the parent process of the calling process is not executing a **wait** system call, and if the parent hasn't set its **SIGCLD** signal to **SIG_IGN**, then the calling process is transformed into a zombie process. A ***zombie process*** is a process that occupies a slot in the process table, but has no other space allocated to it either in user or kernel space. The process table slot that it occupies is partially overlaid with time accounting information to be used by the **times** system call. (See "times" on page 2-165 and the **sys/proc.h** header file.)
- The parent process ID of all of the calling process's existing child processes and zombie processes is set to 1. This means the initialization process inherits each of these processes.
- Each attached shared memory segment is detached and the value of **shm_nattach** in the data structure associated with its shared memory identifier is decremented by 1.
- For each semaphore for which the calling process has set a **semadj** value, that **semadj** value is added to the **semval** of the specified semaphore. (See "semop" on page 2-122 about semaphore operations.)
- If the process has a process lock, text lock, or data lock, an **unlock** is performed. (See "plock" on page 2-97.)

- An accounting record is written on the accounting file if the system's accounting routine is enabled. (See "acct" on page 2-11 for information about enabling accounting routines.)
- If the process ID, tty group ID, and process group ID of the calling process are equal, then the **SIGHUP** signal is sent to each process that has a process group ID equal to that of the calling process. In other words, if **exit** is called by the process group leader for the controlling terminal (typically the shell), then **SIGHUP** is sent to all of the processes associated with that terminal.
- Locks set by the **lockf** system call are removed. (See "lockf" on page 2-64 about file locks.)

The **exit** subroutine causes cleanup actions to occur before the process exits. The **_exit** system call bypasses all cleanup.

Note: The effect of **exit** can be modified by the setting of the **SIGCLD** signal in the parent process. See "signal" on page 2-145 and "sigvec" on page 2-156.

Related Information

In this book: "acct" on page 2-11, "signal" on page 2-145, "sigvec" on page 2-156, "times" on page 2-165, and "wait" on page 2-182.

fclear

Purpose

Makes a hole in a file.

Syntax

long fclear (*fildes***,** *nbytes***)**
int *fildes***;**
unsigned long *nbytes***;**

Description

The **fclear** system call zeroes the number of bytes specified by the *nbytes* parameter starting at the current position of the file open on file descriptor *fildes.* If Distributed Services is installed on your system, this file can reside on another node. This function differs from the logically equivalent write operation in that it returns full blocks of binary zeros to the file system, constructing ***holes*** in the file. The seek pointer of the file is advanced by *nbytes.*

If you **fclear** past the end of a file, then rest of the file is cleared and the seek pointer is advanced by *nbytes.* The file size is updated to include this new hole, which leaves the current file position at the byte immediately beyond the new end-of-file.

Return Value

Upon successful completion, a value of *nbytes* is returned. If the **fclear** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fclear** system call fails if one or more of the following are true:

EIO	I/O error.
EBADF	The *fildes* option is not a valid file descriptor open for writing.
EINVAL	The file is a FIFO, directory, or special file.
EMFILE	The file is mapped copy-on-write by one or more processes.

EAGAIN The write operation in **fclear** failed, due to an enforced write lock on the file.

If Distributed Services is installed on your system, **fclear** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "ftruncate" on page 2-50.

fcntl

Purpose

Controls open file descriptors.

Syntax

#include <fcntl.h>

int fcntl (*fildes*, *cmd*, *arg***)**
int *fildes*, *cmd*, *arg*;

Description

The **fcntl** system call performs controlling operations on open file descriptors. If Distributed Services is installed on your system, the open file can reside on another node.

The *fildes* parameter is an open file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call. The *arg* parameter is a variable that depends on the value of the *cmd* parameter.

The following *cmd*s get a file descriptor or associated flags or set those flags:

F_DUPFD Returns a new file descriptor as follows:

- Lowest numbered available file descriptor greater than or equal to *arg*
- Same open file (or pipe) as the original file
- Same file pointer as the original file (that is, both file descriptors share one file pointer)
- Same access mode (read, write or read/write)
- Same locks
- Same file status flags (that is, both file descriptors share the same file status flags)
- The **close-on-exec** flag associated with the new file descriptor is set to remain open across **exec** system calls.

F_GETFD Gets the **close-on-exec** flag associated with the file descriptor *fildes*. If the low-order bit is 0 (zero), then the file remains open across **exec** system calls; otherwise the file closes upon execution of an **exec** system call.

F_SETFD Sets the **close-on-exec** flag associated with the *fildes* parameter to the value of the low-order bit of *arg* (0 or 1 as for **F_GETFD**).

F_GETFL Gets the file status flags of the file descriptor *fildes.*

F_SETFL Sets the file status flags to the value of the *arg* parameter. Only the flags **O_NDELAY** and **O_APPEND** should be set. Attempting to set other flags may cause unexpected results.

When using the file locking and unlocking *cmd*s (**F_GETLK**, **F_SETLK**, and **F_SETLKW**), the *arg* parameter is a pointer to a structure of type **flock**. The **flock** structure pointed to by the *arg* parameter describes the lock and is defined in the **fcntl.h** header file. It contains the following members:

```
short l_type;              /* F_RDLCK, F_WRLCK, F_UNLCK */
short l_whence;            /* flag for starting offset */
long  l_start;             /* relative offset in bytes */
long  l_len;               /* if 0 then until EOF */
unsigned long l_sysid;     /* node ID */
short l_pid;               /* returned with F_GETLK */
```

l_type Describes the type of lock. Possible values are **F_RDLCK**, **F_WRLCK**, and **F_UNLCK**.

l_whence Defines the starting offset. Possible values of 0, 1, or 2 indicate that the relative offset, **l_start** will be measured from the start of the file, current position, or the end of the file, respectively. Determines the starting point of the relative offset, **l_start**. A value of 0 indicates the start of the file, 1 selects the current position, and 2 indicates the end of the file.

l_start Defines the relative offset in bytes, measured from the starting point in **l_whence**.

l_len Specifies the number of consecutive bytes to be locked.

l_sysid Contains the ID of the node that already has a lock placed on the area defined by the **fcntl** system call. This field is returned only when the **F_GETLK** *cmd* is used.

l_pid Contains the ID of a process that already has a lock placed on the area defined by the **fcntl** system call. This field is returned only when the **F_GETLK** *cmd* is used.

The following *cmd*s use the **flock** structure and perform operations associated with file locks:

F_GETLK Gets the first lock that blocks the lock described in the **flock** structure pointed to by *arg*. If a lock is found, the retrieved information overwrites the information in this structure. If no lock is found that would prevent this lock from being created, then the structure is passed back unchanged except that the lock type is set to **F_UNLCK**.

F_SETLK Sets or clears a file lock according to the **flock** structure pointed to by *arg*. **F_SETLK** is used to establish read (**F_RDLCK**) and write (**F_WRLCK**) locks, as well as to remove either type of lock (**F_UNLCK**). **F_RDLCK**, **F_WRLCK**, and **F_UNLCK** are defined by the **fcntl.h** header file. If a read or write lock cannot be set, **fcntl** returns immediately with an error value of -1.

F_SETLKW Works like **F_SETLK** except that if a read or write lock is blocked by existing locks, the process sleeps until the section of the file is free to be locked.

When a read lock has been set on a section of a file, other processes may also set read locks on that section or subsets of it. A read lock prevents any other process from setting a write lock on any part of the protected area. The file descriptor on which a read lock is being placed must have been opened with read access.

A write lock prevents any other process from setting a read lock or a write lock on any part of the protected area. Only one write lock and no read locks may exist for a specific section of a file at any time. The file descriptor on which a write lock is being placed must have been opened with write access.

Locks may start and extend beyond the current end of a file, but may not be negative relative to the beginning of the file. A lock may be set to extend to the end of the file by setting **l_len** to 0. If such a lock also has **l_start** and **l_whence** set to 0, the whole file will be locked.

Some general rules about file locking include:

- Changing or unlocking part of a file in the middle of a locked section leaves two smaller sections locked at each end of the originally locked section.
- When the calling process holds a lock on a file, that lock is replaced by later calls to **fcntl**.
- All locks associated with a file for a given process are removed when a file descriptor for that file is closed by the process or the process holding the file descriptor ends.
- Locks are not inherited by a child process after executing a **fork** system call.

Notes:

1. In addition to **fcntl**, the **lockf** system call can also be used to set write (exclusive) locks.
2. Deadlocks due to file locks in a distributed system are not always detected. When such deadlocks are possible, the programs requesting the locks should set timeout timers.

Return Value

Upon successful completion, the value returned depends on the value of the *cmd* parameter as follows:

cmd	**Return Value**
F_DUPFD	A new file descriptor
F_GETFD	The value of the flag (only the low-order bit is defined)
F_GETLK	A value other than -1
F_SETFD	A value other than -1
F_GETFL	The value of file flags
F_SETFL	A value other than -1
F_SETLK	A value other than -1
F_SETLKW	A value other than -1.

If the **fcntl** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fcntl** system call fails if one or more of the following are true:

EBADF The *fildes* parameter is not a valid open file descriptor.

EMFILE The *cmd* parameter is **F_DUPFD** and 200 file descriptors are currently open.

EACCES The *cmd* parameter is **F_SETLK**, the *l_type* parameter is **F_RDLCK**, and the segment of the file to be locked is already write-locked by another process.

EACCES The *cmd* parameter is **F_SETLK**, the *l_type* parameter is **F_WRLCK**, and the segment of a file to be locked is already read-locked or write-locked by another process.

Note: Because in the future **errno** may be set to **EAGAIN** rather than to **EACCES** for the two errors described above, programs should expect and test for both values.

EDEADLK The *cmd* parameter is **F_SETLKW**, the lock is blocked by some lock from another process. Putting the calling process to sleep while waiting for that lock to become free would cause a deadlock.

ENOLCK The *cmd* parameter is **F_SETLK** or **F_SETLKW**, the type of lock is **F_RDLCK** or **F_WRLCK**, and there are no more file locks available. (Too many segments are already locked.)

EINVAL The *cmd* parameter is **F_GETLK**, **F_SETLK**, or **F_SETLKW** and the *arg* parameter or the data it points to is not valid.

EINVAL	The *cmd* parameter is **F_DUPFD** and the *arg* parameter is negative or greater than 199.

If Distributed Services is installed on your system, **fcntl** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EAGAIN	The server is too busy to accept the request.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "close" on page 2-25, "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "lockf" on page 2-64, "open" on page 2-90, and "fcntl.h" on page 5-56.

fork

Purpose

Creates a new process.

Syntax

int fork ( )

Description

The **fork** system call creates a new process. The new process (child process) is an exact copy of the calling process (parent process). The created child process inherits the following attributes from the parent process:

- Environment
- Close-on-exec flags (see "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34)
- Signal handling settings (that is, **SIG_DFL**, **SIG_IGN**, *function address*)
- Set-user-ID mode bit
- Set-group-ID mode bit
- Profiling on/off status
- Nice value (see "nice" on page 2-88)
- All attached shared libraries (see **shlib** command in *AIX Operating System Commands Reference*)
- Process group ID
- TTY group ID (see "exit, _exit" on page 2-40 and "signal" on page 2-145)
- Current directory
- Root directory
- File mode creation mask (see "umask" on page 2-169)
- File size limit (see "ulimit" on page 2-167)
- Attached shared memory segments (see "shmat" on page 2-131)
- Attached mapped file segments (see "shmat" on page 2-131).

The child process differs from the parent process in the following ways:

- The child process has a unique process ID.

- The child process has as its parent process ID the process ID of the parent process.

- The child process has its own copy of the parent's file descriptors. However, each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent process.

- All **semadj** values are cleared. (For information about **semadj** values, see "semop" on page 2-122.)
- Process locks, text locks and data locks are not inherited by the child. (For information about locks, see "plock" on page 2-97.)
- The child process's trace flag (see the discussion of request 0 of "ptrace" on page 2-102) is false regardless of the value of the parent process's trace flag.
- The child process's **utime**, **stime**, **cutime**, and **cstime** are set to 0. (See "times" on page 2-165.)
- Any pending alarms are cleared in the child. (See "alarm" on page 2-13.)

Return Value

Upon successful completion, **fork** returns a value of 0 to the child process and returns the process ID of the child process to the parent process. If **fork** fails, a value of -1 is returned to the parent process, no child process is created, and **errno** is set to indicate the error.

Diagnostics

The **fork** system call fails if one or more of the following are true:

EAGAIN The system-imposed limit on the total number of processes executing would be exceeded.

EAGAIN The system-imposed limit on the total number of processes executing for a single user would be exceeded.

ENOMEM There is not enough space left for this process.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, "nice" on page 2-88, "plock" on page 2-97, "ptrace" on page 2-102, "semop" on page 2-122, "shmat" on page 2-131, "signal" on page 2-145, "sigvec" on page 2-156, "times" on page 2-165, "ulimit" on page 2-167, "umask" on page 2-169, and "wait" on page 2-182.

The **shlib** command in *AIX Operating System Commands Reference.*

fsync

Purpose

Writes changes in a file to permanent storage.

Syntax

int fsync (*fildes***)**
int *fildes*;

Description

The **fsync** system call causes all modified data in the file open on *fildes* to be saved to permanent storage. If Distributed Services is installed on your system, this file can reside on another node. If the file is mapped onto a segment in ***read-write*** mode, then it is saved to permanent storage. If the file is mapped ***copy-on-write***, then the pages of the file that have been changed are saved to permanent storage. Saving to permanent storage is sometimes called a ***commit operation***.

An **fsync** system call can be issued by a process executing at the node on which the file is stored or by a process executing at another node. In either case, the file is written to permanent storage at the node that holds the file.

Return Value

Upon successful completion, **fsync** returns a value of 0. If **fsync** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fsync** system call fails if one or more of the following are true:

EIO	I/O error.
EBADF	*fildes* is not a valid file descriptor open for writing.
EINVAL	The file is a FIFO file, directory, or special file.

If Distributed Services is installed on your system, **fsync** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "sync" on page 2-163.

ftruncate

Purpose

Makes a file shorter.

Syntax

int ftruncate (*fildes*, *length***)**
int *fildes*;
unsigned long *length*;

Description

The **ftruncate** system call removes all data beyond *length* bytes from the beginning of the file that is open on the file descriptor *fildes*. Full blocks are returned to the file system so that they can be used again, and the file size is changed to the value of the *length* parameter. If Distributed Services is installed on your system, this file can reside on another node.

The **ftruncate** subroutine does not modify the seek pointer of the file.

Return Value

Upon successful completion, **ftruncate** returns a value of 0. If **ftruncate** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ftruncate** system call fails if one or more of the following are true:

EIO	I/O error.
EBADF	*fildes* is not a valid file descriptor open for writing.
EINVAL	The file is a directory, FIFO, or special file.
EMFILE	The file is mapped copy-on-write by one or more processes.
EAGAIN	The write operation in **ftruncate** failed due to an enforced write lock on the file.

If Distributed Services is installed on your system, **ftruncate** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EAGAIN	The server is too busy to accept the request.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "fclear" on page 2-42.

fullstat, ffullstat

Purpose

Provides information about a file or path to a file.

Syntax

#include <sys/fullstat.h>

int fullstat (*path*, *cmd*, *buf***)**
char **path*;
int *cmd*;
struct fullstat **buf*;

int ffullstat (*fildes*, *cmd*, *buf***)**
int *fildes*, *cmd*;
struct fullstat **buf*;

Description

The **fullstat** system call obtains information about the file pointed to by the *path* parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable. The **fullstat** system call places the information obtained into a structure pointed to by the *buf* parameter, and the *cmd* parameter specifies both the behavior of **fullstat** and the meaning of *buf.*

Use the **ffullstat** system call to obtain information about an open file pointed to by the *fildes* parameter. The *fildes* parameter is a file descriptor obtained from a successful **open**, **creat**, **dup**, **fcntl**, or **pipe** system call. The **ffullstat** system call places the information obtained into a structure pointed to by the *buf* parameter.

The **fullstat** and **ffullstat** system calls provide all of the information available with a **stat** or **fstat** call, plus additional information on group IDs, user IDs, and file location.

The **fullstat** structure pointed to by the *buf* parameter is defined in the **sys/fullstat.h** header file, and it contains the following members:

```
dev_t     st_dev;             /* ID of the device that contains */
                              /*   a directory entry for this file */
ino_t     st_ino;             /* File serial number */
ushort    st_mode;            /* File access mode */
short     st_nlink;           /* Number of links */
ushort    st_uid;             /* Translated UID of the file's owner */
ushort    st_gid;             /* Translated GID of the file's group */
dev_t     st_rdev;            /* ID of device */
```

```
off_t     st_size;          /* File size in bytes */
time_t    st_atime;         /* Time of last access */
time_t    st_mtime;         /* Time of last data modification */
time_t    st_ctime;         /* Time of last file status change */
                            /* Times are measured in seconds since */
                            /*   00:00:00 GMT, Jan. 1, 1970 */
          /* Above fields are the same as stat fields */
ushort    fst_uid_raw;      /* Untranslated user ID of the owner */
                            /*   of the file */
ushort    fst_gid_raw;      /* Untranslated group ID of the file */
vtype     fst_type;         /* Vnode type */
tagtype   fst_uid_rev_tag;  /* Contains special value representing */
                                  /* result of reverse UID mapping */
tagtype   fst_gid_rev_tag;  /* Contains special value representing */
                                  /* result of reverse GID mapping */
short *   fst_other_gid_list;  /* Pointer to first group ID on the */
                               /*   alternate concurrent group list */
short     fst_other_gid_count; /* Number of group IDs on the
                               /*   alternate concurrent group list */
long      fst_vfs;          /* Virtual file system ID */
long      fst_nid;          /* Node ID where the file resides */
int       fst_flag;         /* Indicates whether the directory or */
                            /* file is a virtual mount point */
long      fst_i_gen;        /* Inode generation number */
long      fst_reserved[8];  /* Reserved */
```

st_dev The device that contains a directory entry for this file. On a nondistributed file system, this is a 32-bit quantity that uses only the low 16-bits to contain the concatenated 8-bit major device number and the 8-bit minor device number. On a distributed system, this is a 32-bit quantity, made by combining a 16-bit connection ID, the 8-bit major device number, and the 8-bit minor device number.

st_mode The access mode of the file. (See "stat.h" on page 5-69 for a list of values for this field.)

st_uid The user ID of the file's owner after reverse translation. (See *Managing the AIX Operating System* for a complete discussion of reverse translation.) If the file is a remote file, this value can also be one of the two special values **netnoone** or **netsomeone**, as defined in the **/etc/master** file.

When *cmd* is **FS_STAT_OTHER**, this field is used to input a user ID.

st_gid The group ID of the file's owner after reverse translation. (See *Managing the AIX Operating System* for a complete discussion of reverse translation.) If the file is a remote file, this value can also be one of the two special values **netnoone** or **netsomeone**, as defined in the **/etc/master** file.

When *cmd* is **FS_STAT_OTHER**, this field is used to input a group ID.

st_rdev The ID of the device. This field is defined only for block or character special files.

st_atime The time when file data was last accessed. For remote files, this field contains the time at the server. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **read**.

st_mtime The time when data was last modified. For remote files, this field contains the time at the server. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **write**.

st_ctime The time when file status was last changed. For remote files, this field contains the time at the server. Changed by the following system calls: **chmod**, **chown**, **creat**, **link**, **mknod**, **pipe**, **unlink**, **utime**, and **write**.

fst_uid_rev_tag
For a local file, this field is undefined. For a remote file, this tag describes how **st_uid** was mapped to a remote user ID. Possible values are:

CALLER The **st_uid** returned is that of the calling process, which maps to the file's owner ID.

OTHER The **st_uid** returned is that of a user at this node, which maps to the file's owner ID.

SOMEONE The **st_uid** field is undefined, but some user ID at this node maps to the file's owner ID.

NO_ONE The **st_uid** field is undefined, and no user ID at this node maps to the file's owner ID.

fst_gid_rev_tag
For a local file, this field is undefined. For a remote file, this tag describes how **st_gid** was mapped to a remote group ID. Possible values are:

CALLER The **st_gid** returned is that of the calling process, which maps to the file's group ID.

OTHER The **st_gid** returned is that of a group at this node, which maps to the file's group ID.

SOMEONE The **st_gid** field is undefined, but some group ID at this node maps to the file's group ID.

NO_ONE The **st_gid** field is undefined, and no group ID at this node maps to the file's group ID.

fst_gid_list A pointer to an array that holds an alternate concurrent group list.

fst_gid_count
The number of group IDs on the alternate concurrent group list.

fst_vfs Virtual file system ID. A value of zero indicates the root file system.

fst_flag A flag indicating whether the file or directory is a virtual mount point. A value of **FS_VMP** indicates that it is a virtual mount point.

The following *cmd*s are available:

FL_STAT Returns all of the elements in the structure returned by the **stat** system call, plus additional information about the file. Differs from **FL_STAT_REV** and **FL_STAT_OTHER** in that the values in the **st_uid** and **st_gid** fields are undefined. For this *cmd*, the relevant user ID and group ID information is instead contained in the **fst_uid_raw** and **fst_gid_raw** fields.

This *cmd* allows programs that don't care about the ID values to use the **fullstat** system call without the performance cost of using the translate tables.

FL_STAT_REV Returns all of the elements in the structure returned by the **stat** system call, plus additional information about user and group IDs of the file. When using **FL_STAT_REV**, the returned ID information is the result of the reverse mapping of the user ID and group ID of the calling process. (See *Managing the AIX Operating System* for a complete discussion of reverse translation.)

FL_STAT_OTHER Returns all of the elements in the structure returned by the **stat** system call, plus additional information about the user and group IDs of the file. Differs from **FL_STAT_REV** in that the returned ID information is the result of the reverse mapping of the user ID, group ID, and concurrent group list contained in the **st_uid**, **st_gid**, **fst_other_gid_list**, and **fst_other_gid_count** fields. (See *Managing the AIX Operating System* for a complete discussion of reverse translation.)

When the **FL_STAT_OTHER** *cmd* is used, the return values of **fst_uid_rev_tag** and **fst_gid_rev_tag** are relative to the supplied IDs, rather than to the calling process's IDs. If either **fst_other_gid_list** is **NULL** or **fst_other_gid_count** is 0, there is no concurrent group list.

Return Value

Upon successful completion, both the **fullstat** and the **ffullstat** system calls return a value of 0. If either the **fullstat** or the **ffullstat** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **fullstat** system call fails if one or more of the following are true:

EFAULT The *buf* or *path* parameter points to a location outside of the process's allocated address space.

ENOENT A component of the *path* does not exist.

ENOTDIR A component of the path prefix is not a directory.

EACCES Search permission is denied for a component of the *path.*

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **ffullstat** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

The **ffullstat** system call fails if one or more of the following are true:

EBADF *fildes* is not a valid file descriptor.

EFAULT The *buf* parameter points to a location outside of the process's allocated address space.

If Distributed Services is installed on your system, **ffullstat** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "chown, chownx" on page 2-21, "creat" on page 2-27, "link" on page 2-62, "mknod" on page 2-69, "pipe" on page 2-95, "read, readx" on page 2-106, "stat, fstat" on page 2-159, "time" on page 2-164, "unlink" on page 2-174, "ustat" on page 2-178, "utime" on page 2-180, "write, writex" on page 2-184, "master" on page 4-98, "stat.h" on page 5-69, and "fullstat.h" on page 5-56.2.

Managing the AIX Operating System.

getgroups

Purpose

Gets the group access list.

Syntax

#include <grp.h>

int getgroups (*ngroups*, *gidset***)**
int *ngroups*, **gidset*;

Description

The **getgroups** system call gets the current group access list of the user process. The list is stored in the array pointed to by the *gidset* parameter. The *ngroups* parameter indicates the number of entries that can be stored in this array. **getgroups** never returns more than **NGROUPS** entries. (**NGROUPS** is a constant defined in the **grp.h** header file.)

Return Value

Upon successful completion, the **getgroups** system call returns the number of elements stored into the array pointed to by the *gidset* parameter. If **getgroups** fails, then a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **getgroups** system call fails if the following is true:

EFAULT The *ngroups* and *gidset* parameters specify an array that is partially or completely outside of the process's allocated address space.

Related Information

In this book: "setgroups" on page 2-126 and "initgroups" on page 3-230.

getpid, getpgrp, getppid

Purpose

Gets the process, process group, and parent process IDs.

Syntax

int getpid ( )

int getpgrp ( )

int getppid ( )

Description

The **getpid** system call returns the process ID of the calling process.

The **getpgrp** system call returns the process group ID of the calling process.

The **getppid** system call returns the process ID of the calling process's parent process.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fork" on page 2-46, "setpgrp" on page 2-128, and "signal" on page 2-145.

getuid, geteuid, getgid, getegid

Purpose

Gets the real user, effective user, real group, and effective group IDs.

Syntax

unsigned short getuid ( )

unsigned short geteuid ( )

unsigned short getgid ( )

unsigned short getegid ( )

Description

The **getuid** system call returns the real user ID of the calling process.

The **geteuid** system call returns the effective user ID of the calling process.

The **getgid** system call returns the real group ID of the calling process.

The **getegid** system call returns the effective group ID of the calling process.

Related Information

In this book: "setuid, setgid" on page 2-129.

ioctl

Purpose

Controls input/output devices.

Syntax

#include <sys/ioctl.h>
#include <sys/devinfo.h>

int ioctl (*fildes*, *op*, *arg*)
int *fildes*, *op*;
char **arg*;

Description

The **ioctl** system call performs a variety of control operations on the block or character special file (device) specified by the *fildes* parameter. The *op* parameter specifies the operation, and the use of the *arg* parameter depends on the particular operation performed. The **ioctl** operations that are valid for each type of device are explained in Chapter 6, "Special Files."

Two operations are valid for all types of devices that supports **ioctl** system call. These two operations are:

IOCTYPE Returns the device type associated with *fildes*. The device types are defined in the **sys/devinfo.h** header file, which is discussed in "devinfo" on page 4-57.

IOCINFO Stores device information for the file specified by *fildes* into the buffer pointed to by the *arg* parameter. See "devinfo" on page 4-57 for the format of the device information structure.

Some devices support additional requests. See the discussion of individual devices in Chapter 6, "Special Files" for details about device-dependent **ioctl** calls.

Return Value

If **ioctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ioctl** fails if one or more of the following are true:

EBADF	*fildes* is not a valid open file descriptor.
ENOTTY	*fildes* is not associated with a character special file.
ENODEV	The device associated with *fildes* does not support the **ioctl** system call.
EFAULT	The *arg* parameter points to a location outside of the process's allocated address space.
EINVAL	*op* or *arg* is not valid.
EINTR	A signal was caught during the **ioctl** system call.

Related Information

In this book: "devinfo" on page 4-57, Chapter 6, "Special Files," Appendix C, "Writing Device Drivers," and "*dd***ioctl**" on page C-8 .

The discussion of **termio** in *AIX Operating System Programming Tools and Interfaces.*

iplvm, waitvm

Purpose

Starts a virtual machine or waits for one to terminate.

Syntax

int iplvm (*iodn***,** *waitflag***)**
unsigned short *iodn***;**
int *waitflag***;**

int waitvm (*iodn***)**
unsigned short *iodn***;**

Description

The **iplvm** system call starts (IPLs) a new virtual machine that is independent of the AIX virtual machine. The virtual machine is loaded from the device specified by the input/output device number (IODN) given in the *iodn* parameter.

If *iodn* is 0, then the virtual machine of the calling process is restarted (re-IPLed). This is accomplished by sending a **SIGQUIT** signal to the process 1. (Process 1 is also called the **init** process. See "Creation and Execution" on page 1-16 for more information about this special process.) Note that, unlike the **reboot** system call, **iplvm** performs a **sync** operation and writes all pending output to disk before restarting the virtual machine.

If the *waitflag* parameter is a nonzero value, then the **iplvm** system call waits until the new virtual machine has started before returning to the calling process. This allows you to determine whether the virtual machine IPLed successfully. *waitflag* is ignored if the *iodn* parameter is 0.

The **waitvm** system call waits for the virtual machine that was IPLed from the device specified by *iodn* to perform a virtual machine halt.

The calling process must have an effective user ID of superuser to perform either system call.

Return Value

Upon successful completion, the **iplvm** and **waitvm** system calls return a value of 0. If **iplvm** or **waitvm** fails, then a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **iplvm** and **waitvm** system calls fail if one or more of the following is true:

EPERM The effective user ID of the calling process is not superuser.

EIO The VRM detected one of the following error conditions:

- Insufficient resources are available.
- The IPL diskette needs to be mounted.
- The IPL header is not valid.
- Error encountered while reading IPL record.
- Duplicate virtual machine ID.
- IPL key sequence error.

ENXIO The device specified by the *iodn* parameter does not exist.

EINVAL The *iodn* parameter is not valid.

EAGAIN The maximum number of virtual machines are already running.

Related Information

In this book: "Creation and Execution" on page 1-16, "reboot" on page 2-109, "signal" on page 2-145, and "sync" on page 2-163.

The **init** command in *AIX Operating System Commands Reference.*

kill

Purpose

Sends a signal to a process or to a group of processes.

Syntax

int kill (*pid*, *sig***)**
int *pid*, *sig*;

Description

The **kill** system call sends the signal specified by the *sig* parameter to the process or group of processes specified by the *pid* parameter. (For information on valid signals, see "signal" on page 2-145.) If the *sig* parameter is 0 (the ***null signal***), error checking is performed but no signal is sent. This can be used to check the validity of *pid*.

To send a signal to another process, at least one of the following must be true:

- Either the real or the effective user ID of the sending process matches the real or effective user ID of the receiving process.
- The effective user ID of the sending process is superuser.

The processes that have the process IDs 0 and 1 are special processes and are sometimes referred to here as ***proc0*** and ***procl***, respectively.

If the *pid* parameter is greater than 0, the signal specified by the *sig* parameter is sent to the process whose process ID is equal to the value of the *pid* parameter.

If the *pid* parameter is equal to 0, the signal specified by the *sig* parameter is sent to all of the processes, excluding **proc0** and **procl**, whose process group ID is equal to the process group ID of the sender.

If the *pid* parameter is equal to -1 and the effective user ID of the sender is not superuser, the signal specified by the *sig* parameter is sent to all of the processes, excluding **proc0** and **procl**, whose real user ID is equal to the effective user ID of the sender.

If the *pid* parameter is equal to -1 and the effective user ID of the sender is superuser, the signal specified by the *sig* parameter is sent to all of the processes, excluding **proc0** and **procl**.

If the *pid* parameter is negative but not -1, the signal specified by the *sig* parameter is sent to all of the processes whose process group ID is equal to the absolute value of the *pid* parameter.

Return Value

Upon successful completion, **kill** returns a value of 0. If **kill** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **kill** system call fails and no signal is sent if one or more of the following are true:

EINVAL *sig* is not a valid signal number.

EINVAL *sig* is **SIGKILL** and *pid* is 1 (**procl**).

ESRCH No process can be found corresponding to that specified by *pid*.

EPERM The user ID of the sending process is not superuser, and the real or effective user ID does not match the real or effective user ID of the receiving process.

Related Information

In this book: "getpid, getpgrp, getppid" on page 2-54, "setpgrp" on page 2-128, and "signal" on page 2-145.

The **kill** command in *AIX Operating System Commands Reference.*

link

Purpose

Creates an additional directory entry for an existing file.

Syntax

int link (*path1***,** *path2***)**
char ****path1*, ****path2***;**

Description

The **link** system call creates an additional link (directory entry) for an existing file. The *path1* parameter points to the the path name of an existing file and the *path2* parameter points to the path name for the new directory entry to be created. If Distributed Services is installed on your system, these paths can cross into another node.

Return Value

Upon successful completion, **link** returns a value of 0. If **link** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **link** system call fails if one or more of the following are true:

ENOTDIR A component of either path prefix is not a directory.

ENOENT A component of either path prefix does not exist.

EACCESS A component of either path prefix denies search permission.

ENOENT The file named by the *path1* parameter does not exist.

EEXIST The link named by the *path2* parameter already exists.

EPERM The file named by the *path1* parameter is a directory and the effective user ID is not superuser.

EXDEV The link named by the *path2* parameter and the file named by the *path1* parameter are on different file systems.

ENOENT The *path2* parameter points to a null path name.

EACCES The requested link requires writing in a directory with a mode that denies write permission.

EROFS The requested link requires writing in a directory on a read-only file system.

EFAULT The *path1* or *path2* parameter points to a location outside of the process's allocated address space.

EMLINK The file already has the maximum number of links.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **link** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "unlink" on page 2-174.

The **link** command in *AIX Operating System Commands Reference.*

loadtbl

Purpose

Installs or queries configuration information in the kernel.

Syntax

#include <sys/dstables.h>

int loadtbl (*cntl*, *buf*, *size*)
**struct ltable ****cntl*;
**char ****buf*;
int *size*;

Description

The **loadtbl** system call installs one element of a table into the kernel or queries information from the kernel. If Distributed Services is installed on your system, you can query, but not change, the kernel of another node.

To change or add an entry in the kernel table with the **loadtbl** system call, a process must have an effective user ID of superuser. Queries of table entries can be submitted by any process.

This system call provides a general mechanism for installing configuration information into the kernel. The outline of the general structure supported by **loadtbl** is:

- For each **type**, there is an array of table headers that has a fixed size.
- Each table header contains an ID and a pointer to a structure of information related to that ID.
- The structure of *buf* is unknown to the **loadtbl** system call.

The **loadtbl** system call can be used to load a single table entry into the kernel, to load an entire set of table entries into the kernel, or to query one or more entries in the kernel.

The *buf* parameter is a pointer to the data being loaded, while the *size* parameter determines the size (in bytes) of the data being loaded. The *cntl* parameter points to a structure of type **ltable**. The **ltable** structure is defined in the **sys/dstables.h** header file, and it contains the following members:

```
      char type;     /* Defines the type of table being managed   */
                     /*   or queried.  The type must be one of:   */
                     /*    DSNCB   (Node Control Block)           */
                     /*    DSOCB   (Outbound Translate)           */
                     /*    DSWCB   (Wild Card Node Translate)     */
                     /*    DSIPC   (IPC key table entries)        */
                     /*    USERTBL (User-defined Table)           */
      long id;       /* Identifies which table entry is being     */
                     /*   loaded.  For DSNCB this is the node ID. */
      char mode;     /* Type of request.                          */
      long nid;      /* For queries, the node ID whose tables are */
                     /*   to be queried.                          */
      long reserved[4];  /* Reserved.  Contents must be zero      */
```

The **mode** field of the **ltable** structure can contain any one of the following values:

L_REPLACE The entry being loaded is either a new one or a replacement for an existing one. If no corresponding table entry exists, a new one is created.

L_DELETE The table information for the **id** specified is deleted.

L_QUERYI This mode enables the calling process to learn which IDs of a particular type currently have information in the kernel. The **ltable.type** field specifies which type is being queried. *buf* is assumed to point to an array of long integers into which **loadtabl** places the requested IDs. (When **type** is **DSOCB**, **DSWCB**, or **DSIPC**, a single zero ID is returned in the array.) The **ltable.nid** specifies which node's kernel is to be queried. If **ltable.nid** is 0 or the node ID of the local node, then the local kernel is queried. Otherwise, the specified remote node ID is queried.

L_QUERYT This mode enables the calling process to learn the table values that currently reside in the kernel for a particular table **type** and **id**. The **ltable.type** field specifies which type of table is being queried, and **ltable.id** specifies which ID is being queried. The **ltable.nid** field specifies which node's kernel is to be queried. If **ltable.nid** is 0 or the node ID of the local node, then the local kernel is queried. Otherwise, the specified remote node ID is queried. *buf* is assumed to point to an array of long integers into which **loadtabl** places the requested information.

Three of the **ltable.type**s, **DSNCB**, **DSOCB**, and **DSWCB**, are associated with ID translates. The fourth type, **DSIPC**, is used when an IPC key mapping table is being loaded. The fifth type is **USERTBL**, which is used to load tables for international character support into the kernel. The following sections describe the ID translate values first, then the key map information, and, finally, the international character support tables.

The ID translate **type**s are:

DSNCB An inbound user ID or group ID translate table for a particular node.

DSOCB A translate table for outbound requests.

DSWCB A translate table for the wild card node.

In all of these cases, the memory pointed to by *buf* contains a **dsxlate** structure, which contains, among other things, the first row of translate information, followed by the rest of the translate rows. For **DSNCB**, the **ltable.id** specifies which node ID's table is used.

The **dsxlate** structure is described in the **sys/dstables.h** header file, and it contains the following members:

```
short rlvl;      /* Reserved.  Must be 0 - release level.      */
short gid;       /* Local wildcard group ID for this node.     */
short uid;       /* Local wildcard user ID for this node.      */
char flag;       /* The 0X01 bit is set if there is a wildcard */
                 /*   user ID for this node.                   */
                 /* The 0X02 bit is set if there is a wildcard */
                 /*   group ID for this node.                  */
char pad1;       /* Dummy for alignment purposes               */
short numuids;   /* The number of user ID translate rows       */
short numgids;   /* The number of group ID translate rows      */
short pad2;      /* Dummy for alignment purposes               */
struct idrow idrow[1];  /* First row of translate information  */
                         /*   Rest of the translates follow.   */
```

The **idrow** structure is described in the **sys/dstables.h** header file, and it contains the following members:

```
long wireid;     /* ID that arrived with the request           */
short localid;   /* Local ID resulting from a translate        */
short pad;       /* Dummy for alignment purposes               */
```

Note: For **ltable.type** values of **DSNCB** and **DSWCB**, the entries in the user ID and group ID table arrays must be ordered by increasing **wireid**. For a **type** value of **DSOCB** (outbound translate table), the table must be ordered by increasing **localid**.

There are occasions when an application needs to load an entire set of new translates into the kernel. In the normal processing sequence for such a case, the application should:

1. Call the **loadtbl** system call with `ltable.type=DSNCB` and `ltable.mode=L_QUERYT` to get a list of the node IDs that currently have translates.
2. Compare the returned list of node IDs that currently have translates to the list of node IDs that should have translates after the new information is loaded.
3. Use **L_DELETE** to remove the node IDs that now have translates, but should not.

4. Use **L_REPLACE** on the translates for the other nodes (both node IDs that have existing translates and node IDs that are not yet in the table).

The next **ltable.type** loads a different type of information into the kernel than the ID translate types listed earlier. When **ltable.type** is **DSIPC**, an IPC key mapping table is being loaded. The value of **ltable.id** is ignored for this **type**, and the entire set of IPC keys is loaded as one piece of information. The memory pointed to by *buf* should be an array of **dsipc** structures. The **dsipc** structure is defined in the **sys/dstables.h** file, and it contains the following members:

```
long inkey;      /*  Input key   */
long nid;        /*  Node ID     */
long outkey;     /*  Result key  */
```

A **dsipc.nid** with a value of 0 indicates that the **inkey** is to be mapped to the **outkey** in the local node. Otherwise, the specified node is used.

The array should be sorted by **inkey** so that the **msgget** subroutine can use a binary search to locate the requested key translate.

The final **ltable.type** is used to load tables of a different nature than described above for Distributed Services. When **ltable.type** is **USERTBL**, the table being loaded into the kernel is used for international character support, such as a character collation and classification table. (See "Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System* for additional information on character collation and classification.)

Return Value

Upon successful completion, the **loadtbl** system call returns:

- For **L_REPLACE**, a value of 0.
- For **L_DELETE**, a value of 0.
- For **L_QUERYI**, a value of 0 if no table exists for the **type** specified. Otherwise, the number of IDs of the specified **type** is returned.
- For **L_QUERYT**, the number of bytes transferred.

If the **loadtbl** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **loadtbl** system call fails and the configuration information remains unchanged if one or more of the following are true:

E2BIG	*buf* is too small to contain query data.
EINVAL	Invalid input data (such as invalid **ltable.type** or **ltable.mode**, or an **ltable.nid** that specifies a remote node when **ltable.mode** is **L_REPLACE** or **L_DELETE**).
EPERM	The **ltable.mode** is either **L_REPLACE** or **L_DELETE** and the effective user ID of the calling process is not superuser.

If Distributed Services is installed on your system, the **loadtbl** system call also fails if one or more of the following are true:

EDIST	Outbound requests are currently blocked.
EDIST	The server has a release level of Distributed Services that cannot communicate with this node.
EAGAIN	The server is too busy to accept the request.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
ENOCONNECT	An attempt to establish a new network connection with a remote node failed.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "msgget" on page 2-76.

The **dsldxprof** command in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

Managing the AIX Operating System.

AIX Operating System Programming Tools and Interfaces.

lockf

Purpose

Locks a region of a file for exclusive access.

Syntax

#include <sys/lockf.h>

int lockf (*fildes***,** *request***,** *size***)**
int *fildes***,** *request***;**
off_t *size***;**

Description

The **lockf** system call locks and unlocks regions of an open file. If Distributed Services is installed on your system, this file can reside on another node. **lockf** is used to synchronize simultaneous access to a file by multiple processes. Only one process at a time can hold a lock on any given region of a file. Two types of locks are provided: enforced and advisory.

When a process holds an ***enforced lock*** on a region of a file, no other process can access that region with the **read** or **write** system calls. In addition, **creat** and **open** are prevented from truncating the file. If another process attempts to **read** or **write** the region, then it sleeps until the region is unlocked. However, if the system detects that sleeping would cause deadlock, then the **read** or **write** system call fails, setting **errno** to **EDEADLK**. If another process attempts to truncate the file with either the **creat** or **open** system call, then that system call fails and sets **errno** to **EACCES**.

When a process holds an ***advisory lock*** on a region of a file, no other process can lock that region, or an overlapping region, with **lockf**. The **read**, **write**, **creat**, and **open** system calls are not affected. This means that processes must voluntarily call **lockf** in order to make advisory locks effective.

Warning: Buffered I/O does not work properly when used with file locking. In particular, do not use the Standard I/O Package **(libc.a)** routines on files that are going to be locked, since these routines use buffered I/O.

To select enforced locking, the **S_ENFMT** bit must be set in the access permission code (or mode) of the file. Otherwise locking is advisory. Thus, a given file can have advisory or enforced locks, but not both.

The *fildes* parameter to **lockf** is an open file descriptor obtained from a successful call to **open**, **creat**, **dup**, or **pipe** system call.

The *size* parameter is the number of bytes to be locked or unlocked. The region starts at the current location in the open file and extends forward if *size* is positive, or backward if *size* is negative. If the *size* parameter is 0, then the region starts at the current location and extends forward to the maximum possible file size, including the unallocated space after the end-of-file. Unallocated "holes" in the file can also be locked. (See "fclear" on page 2-42 about "holes" in files.)

The *request* parameter is one of the following constants:

F_ULOCK Unlocks a previously locked region in the file.

F_LOCK Locks the region for exclusive use. This *request* causes the calling process to sleep if the region overlaps a locked region, and to resume when it is granted the lock.

F_TLOCK Tests to see if another process has locked the specified region, and, if not, locks the region for exclusive use. If the region is already locked, then **lockf** fails and sets **errno** to **EACCES**.

F_TEST Tests to see if another process has already locked a region. **lockf** returns 0 if the region is unlocked. If the region is locked, then -1 is returned and **errno** is set to **EACCES**.

The system keeps a table of the locked regions for each file. This table can hold a limited number of entries. When the same process locks two regions that are next to each other in the file, **lockf** combines the lock table entries to conserve space in the lock table. An unlock request in the middle of a locked region leaves two locked regions, which can cause the lock table to overflow. When a lock or unlock request cannot be satisfied because the lock table is full, the **lockf** subroutine fails.

When a process closes a file, all of its locks on that file are removed. When a process terminates, all of the locks that it holds are removed.

All locks applied to directories, special files, and pipes are treated as advisory locks. However, locking directories is not recommended. Only advisory locks are supported for mapped files. An attempt to apply an enforced lock to a mapped file causes the **lockf** system call to fail and set **errno** to **EMFILE**. (For information about mapped files, see "shmat" on page 2-131.)

A child process does not inherit the locks of its parent process.

Notes:

1. Locks may be set by **fcntl** in addition to **lockf**.
2. The **lockf** system call sets only write (exclusive) locks.
3. When using Distributed Services, deadlocks due to file locks are not always detected. When such deadlocks are possible, the programs requesting the locks should set timeout timers.

Return Value

Upon successful completion, **lockf** returns a value of 0. If **lockf** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **lockf** system call fails if one or more of the following are true:

EBADF *fildes* is not a valid open file descriptor.

EINVAL *request* is not valid.

EACCES **F_TEST** or **F_TLOCK** fails because another process has already locked the region.

Note: Because in the future, **errno** may be set to **EAGAIN** rather than to **EACCES** for this error described above, programs should expect and test for both values.

EMFILE The file is mapped and enforced locking is enabled.

EDEADLK Deadlock will occur or the lock table is full. Deadlocks are not always detected when remotely mounted files are locked.

If Distributed Services is installed on your system, **lockf** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "close" on page 2-25, "creat" on page 2-27, "dup" on page 2-32, "fcntl.h" on page 5-56, "open" on page 2-90, "read, readx" on page 2-106, "write, writex" on page 2-184, and "standard i/o library" on page 3-342.

lseek

Purpose

Moves read/write file pointer.

Syntax

#include <sys/types.h>
#include <unistd.h>

off_t lseek (*fildes*, *offset*, *whence***)**
int *fildes*;
off_t *offset*;
int *whence*;

Description

The **lseek** system call sets the file pointer for the file specified by the *fildes* parameter. If Distributed Services is installed on your system, this file can reside on another node. The *fildes* parameter is a file descriptor obtained from a **creat, open, dup,** or **fcntl** system call.

The **lseek** system call sets the file pointer associated with the *fildes* stream according to the value of the *whence* parameter, as follows:

SEEK_SET Sets the file pointer to the value of the *offset* parameter.

SEEK_CUR Sets the file pointer to its current location plus the value of the *offset* parameter.

SEEK_END Sets the file pointer to the size of the file plus the value of the *offset* parameter.

Return Value

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned. If **lseek** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **lseek** system call fails and the file pointer remains unchanged if one or more of the following are true:

EBADF *fildes* is not an open file descriptor.

ESPIPE *fildes* is associated with a pipe (FIFO) or a multiplexed special file.

EINVAL *whence* is not 0, 1 or 2. This also causes a **SIGSYS** signal.

EINVAL The resulting file pointer would be negative.

If Distributed Services is installed on your system, **lseek** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "creat" on page 2-27, "dup" on page 2-32, "fcntl" on page 2-44, "open" on page 2-90, and "fseek, rewind, ftell" on page 3-196.

mkdir

Purpose

Creates a directory.

Syntax

int mkdir (*path*, *mode***)**
**char ****path*;
int *mode*;

Description

The **mkdir** system call creates a new directory. The *path* parameter names the new directory. If Distributed Services is installed on your system, this path can cross into another node. In this case, the new directory is created at that node.

To execute the **mkdir** system call, a process must have search permission and write permission in the parent directory of *path.*

The *mode* parameter is the mask for the read, write, and execute (rwx) flags for owner, group, and others. The low-order 9 bits in *mode* are modified by the file mode creation mask of the process. All bits set in the creation mask are cleared. (For more information about the creation mask, see “umask” on page 2-169.)

Return Value

Upon successful completion, the **mkdir** system call returns a value of 0. If the **mkdir** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **mkdir** system call fails and the directory is not created if one or more of the following are true:

ENOTDIR A component of the path is not a directory.

ENOENT A component of the path does not exist.

EACCES Creating the requested directory requires writing in a directory with a mode that denies write permission.

EACCES Search permission is denied for a component of the path.

EROFS The named file resides on a read-only file system.

EEXIST The named file already exists.

EFAULT The *path* parameter points outside of the process's allocated address space.

EIO An I/O error occurred while writing to the file system.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **mkdir** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "mknod" on page 2-69, "rename" on page 2-110.1, "rmdir" on page 2-110.4, and "umask" on page 2-169.

mknod

Purpose

Creates a directory, a special file, or an ordinary file.

Syntax

```
#include <sys/stat.h>

int mknod (path, mode, dev)
char *path;
int mode;
dev_t dev;
```

Description

The **mknod** system call creates a new regular file, special file, or directory. The *path* parameter names the new file. If Distributed Services is installed on your system, this path can cross into another node. Also see "mkdir" on page 2-68.1 for additional information on creating a directory.

The *mode* parameter specifies the ***mode*** of the file, which defines the file type and access permissions.

The *dev* parameter is configuration dependent and is used only if the *mode* parameter specifies a block or character special file. *dev* is the ID of the device, and it corresponds to the **st_rdev** member of the structure returned by the **stat** system call. If the file you specify is a remote file, the *dev* value must be meaningful on the node where the file resides. See "stat, fstat" on page 2-159 and "stat.h" on page 5-69 for more information about the device ID.

The *mode* parameter is constructed logically OR-ing the values specified in "chmod" on page 2-18 with one the following values, which define the file type:

S_IFDIR	Directory
S_IFCHR	Character special file
S_IFMPX	Multiplexed character special file
S_IFBLK	Block special file
S_IFREG	Regular data file
S_IFIFO	FIFO special file

A complete list of the possible *mode* values and other useful macros appears in "stat.h" on page 5-69.

The low-order 9 bits of *mode* are modified by the process's file mode creation mask: all bits set in the creation mask are cleared. (For more information about the creation mask, see "umask" on page 2-169.)

If the type of the new file is **S_IFMPX** (multiplexed character special file), then when the file is used, additional path name components can appear after the path name as if it were a directory. The additional part of the path name is available to the file's device driver for interpretation. This provides a multiplexed interface to the device driver. The **hft** device driver uses this feature. (See "hft" on page 6-23 for details about this device driver.)

The file's owner ID is set to the process's effective user ID. The file's group ID is set to the process's effective group ID.

The **mknod** system call can be invoked only by superuser for file types other than FIFO special.

Return Value

Upon successful completion, a value of 0 is returned. If the **mknod** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **mknod** system call fails and the new file is not created if one or more of the following are true:

EPERM The process's effective user ID is not superuser.

ENOTDIR A component of the path prefix is not a directory.

ENOENT A component of the path prefix does not exist.

EROFS The directory in which the file is to be created is located on a read-only file system.

EEXIST The named file exists.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **mknod** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34,"mkdir" on page 2-68.1 , "umask" on page 2-169, "fs" on page 4-74, and "stat.h" on page 5-69.

The **chmod**, **mkdir**, and **mknod** commands in *AIX Operating System Commands Reference.*

mntctl

Purpose

Returns information about a node's mounts.

Syntax

#include <sys/mntctl.h>

int mntctl (*cmd*, *size*, *buf***)**
int *cmd*, *size*;
char **buf*;

Description

The **mntctl** system call returns information about a node's mounts. The *size* parameter defines the number of bytes in *buf*. The *cmd* parameter specifies how **mntctl** acts and specifies the meaning of *buf*. The following *cmd* is available:

MC_MOUNTS Queries a node to learn what that node has mounted. This can be the local node or, if Distributed Services is installed on your system, a remote node.

The *buf* parameter points to a **bheader** structure that contains, among other things, the node ID of the node to query and, after the system call completes, a description of the first mount issued by that node. The **mntctl** system call updates the **size** parameter and first **minfo** structure in **bheader**, then appends descriptions of additional mounts in separate **minfo** structures. In this way, **bheader.minfo** becomes the first element in an array of **minfo** structures. The strings pointed to by the **m_object** and **m_stub** fields of each **minfo** structure are appended to the buffer after the array of **minfo** structures.

The **bheader** structure pointed to by the *buf* parameter is defined in the **sys/mntctl.h** header file, and it contains the following members:

```
unsigned long nid;    /* the node ID of the node that is being */
                      /* queried.  Zero indicates the local node. */
int reserved;
unsigned int size;    /* the number of minfo structures in the */
                      /* buffer, including the one below in */
                      /* bheader, or the size of buf (in bytes) */
                      /* that would be required to hold the */
```

```
                        /* requested info */
struct minfo[1];        /* first structure of mount info    */
```

The **minfo** structure is defined in the **sys/mntctl.h** file, and it contains the following members:

```
unsigned long m_nid;    /* the node ID of the node that holds the */
                        /*  mounted object.  Zero indicates the */
                        /*  local node. */
char *m_object;         /* path to the mounted object */
char *m_stub;           /* path to the mounted-over object */
unsigned int m_flag;    /* flag bits to define characteristics of the */
                        /*  mounted object    */
time_t m_date           /* date that the virtual file system */
                        /* was created */
```

m_object Defines the path to the object to be mounted. For the root device, this field contains a pointer to a **NULL** string.

m_flag Defines various characteristics of the object to be mounted. Possible values for this field are defined in the **sys/vmount.h** file, which is included by the **sys/mntctl.h** file. These values are:

MNT_READONLY Indicates that the object was mounted with a read-only mount, and write access is not allowed.

MNT_REMOVABLE Indicates that the object is a removable file system.

MNT_REMOTE Indicates that the mounted object resides on a node other than the queried node.

MNT_DEVICE Indicates that the mounted object is a device.

The values returned for **m_object** and **m_stub** are the paths used when a **mount** or **vmount** system call was issued to mount that object. If the paths used with these calls are not full path names, the **minfo** structures returned by the **mntctl** system call may not be very useful.

When the **mntctl** system call completes successfully, the **size** field in the **bheader** structure contains the total number of **minfo** structures, including the one in **bheader**. If *buf* is not big enough to hold all of the **minfo** structures and all of the strings, the **mntctl** system call fails and **bheader.size** contains the size (in bytes) that *buf* must be to hold all of the requested data. If additional mounts are performed at the remote node before the **mntctl** system call is reissued with the new *size* parameter, this error can occur again.

Return Value

Upon successful completion, the **mntctl** system call returns a value of 0. If the **mntctl** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **mntctl** system call fails and the requested information is not returned if one or more of the following are true:

E2BIG The requested information will not fit into *size.*

If Distributed Services is installed on your system, **mntctl** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "mount" on page 2-71, "vmount" on page 2-180.5, and "master" on page 4-98.

mount

Purpose

Mounts a file system.

Syntax

```
#include <sys/vmount.h>

int mount (dev, dir, mflag)
char *dev, *dir;
int mflag;
```

Description

The **mount** system call mounts a file system contained on the block device (also called a ***special file***) identified by the *dev* parameter. The file system is mounted on the directory identified by the *dir* parameter. The **mount** system call can be used only by superuser.

The *dev* parameter and the *dir* parameter are pointers to path names.

The *mflag* parameter defines various characteristics of the object to be mounted. Possible values are:

MNT_READONLY Indicates that the to be mounted object is read-only, and write access is not allowed. If this value is not specified, writing is permitted according to individual file accessibility.

MNT_REMOVABLE Indicates that the object to be mounted is a removable file system. Whenever there are no active references to files or directories on the file system, the operating system forgets the content and structure of the file system. The user can remove the medium and replace it with a different file system. All future references to *dir* will refer to the file system on the new medium.

After the file system is mounted, references to the path name specified by the *dir* parameter refer to the root directory on the mounted file system.

Return Value

Upon successful completion a value of 0 is returned. If the **mount** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **mount** system call fails if one or more of the following are true:

EPERM The effective user ID of the calling process is not superuser.

ENOENT *dev* or *dir* does not exist.

ENOTBLK *dev* is not a block device.

ENXIO The device or driver for *dev* is not currently configured.

ENOTDIR A component of a path prefix is not a directory.

ENOTDIR *dir* is not a directory.

ENOTDIR The path to the device being mounted crosses a remote mount point.

EFAULT The *dev* or *dir* parameter points to a location outside of the process's allocated address space.

EBUSY *dir* is currently busy. For example, it may some process's current directory, or a file system may be mounted onto it.

EBUSY The device associated with *dev* is currently mounted.

EBUSY There are no more mount table entries.

EINVAL The data on *dev* is not recognizable as a file system. This usually means that it does not contain a properly formatted super-block or, if Distributed Services is installed on your system, that *dev* or *dir* is on a remote node.

ENOMEM Either this node or the server does not have enough memory available to service the request.

E2BIG The length of the path pointed to by either the *dev* or *dir* parameter is greater than the value of **MAXPATH**.

Related Information

In this book: "umount" on page 2-170, "uvmount" on page 2-180.3, "vmount" on page 2-180.5, and "fs" on page 4-74.

The **mount** and **umount** commands in *AIX Operating System Commands Reference.*

msgctl

Purpose

Provides message control operations.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (*msqid*, *cmd*, *buf***)**
— or —
int msgctl (*msqid*, *cmd*, *mtype***)**

int *msqid*, *cmd*;
struct msqid_ds **buf*;
int *mtype*;

Description

The **msgctl** system call provides a variety of message control operations as specified by *cmd* parameter. The *buf* parameter points to a structure of type **msqid_ds**. The **msqid_ds** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
struct ipc_perm  msg_perm;    /* Operation permission structure       */
struct msg     *msg_first;   /* Ptr to first message on the queue    */
struct msg     *msg_last;    /* Ptr to last message on the queue     */
ushort          msg_cbytes;  /* Current number of bytes on the queue */
ushort          msg_qnum;    /* Number of messages on the queue      */
ushort          msg_qbytes;  /* Maximum number of bytes on the queue */
ushort          msg_lspid;   /* ID of last process to call msgsnd    */
ushort          msg_lrpid;   /* ID of last process to call msgrcv    */
time_t          msg_stime;   /* Time of last msgsnd call             */
time_t          msg_rtime;   /* Time of last msgrcv call             */
time_t          msg_ctime;   /* Time of the last change to this      */
                             /*  structure with a msgctl call        */
/*      The following members support Distributed Services IPC:     */
long            msg_nid;      /* DS - real queue's node ID          */
int             msg_qid;      /* DS - real queue's queue ID         */
key_t           msg_rkey;     /* DS - caller's mapped key           */
ushort          msg_lsnid;    /* DS - nid of last msgsnd            */
ushort          msg_lrnid;    /* DS - nid of last msgrcv            */
mtyp_t          msg_mtype;    /* DS - server's mtype value          */
int             msg_bootcnt;  /* DS - server's boot count           */
struct node     *msg_conn;    /* DS - connect for this surrogate    */
```

The following *cmd*s are available:

IPC_STAT Stores the current value of the first eleven members of the data structure associated with the *msqid* parameter into the **msqid_ds** structure pointed to by the *buf* parameter. The structure members that support Distributed Services are not included.

The current process must have read permission in order to perform this operation. If the queue resides in a remote node, then the information in the structure (**msg_lspid**, **msg_stime**, and so on) is relative to that node.

If the last sent (or received) message came from a node other than the one that holds the queue, then the process ID (pid) reported for **msg_lspid** is the process ID of the transaction process that actually operated on the queue, not the process ID at the remote machine.

IPC_SET Sets the value of the following members of the data structure associated with the *msqid* parameter to the corresponding values found in the structure pointed to by the *buf* parameter:

```
msg_perm.uid
msg_perm.gid
msg_perm.mode  /* Only the low-order nine bits */
msg_qbytes
```

The current process must have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid* in order to perform this operation. To raise the value of **msg_qbytes**, the effective user ID of the current process must be superuser.

If *msqid* identifies a remote queue, the remote node uses the translated versions of the caller's user and group IDs to determine if the caller has permission to delete the queue. See *Managing the AIX Operating System.*

For remote queues, the user and group IDs of the calling process are sent to the node where the queue resides. The remote node performs inbound ID translation and uses the result to determine if the caller has permission to run the command. See *Managing the AIX Operating System* for a discussion of user and group ID translation.

IPC_RMID Removes the message queue identifier specified by the *msqid* parameter from the system and destroys the message queue and data structure associated with it. The current process must have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid* in order to perform this operation.

If *msqid* identifies a remote queue, the remote node uses the translated versions of the caller's user and group IDs to determine if the caller has permission to delete the queue. See *Managing the AIX Operating System* for a discussion of user and group ID translation.

If a remote queue cannot be removed (for example, if permission is denied), then the local queue header is not removed either. Otherwise, both the remote and local queue information is removed.

If Distributed Services is installed on your system, the following *cmd*s are also available:

IPC_STAT2 Returns information about message queues, including the Distributed Services structure members, regardless of whether the queue is remote or local. *buf* is assumed to point to a **msqid_ds** structure. New programs that wish to learn status information about IPC queues should use **IPC_STAT2** rather than **IPC_STAT**.

IPC_RMID2 Removes the local header for the remote queue associated with *msqid* without attempting to remove the remote queue itself. If *msqid* does not identify a remote queue, then **msgctl** sets **errno** to **EINVAL**. Like **IPC_RMID**, **IPC_RMID2** requires that the current process have an effective user ID equal to either that of superuser or to the value of **msg_perm.uid** in the data structure associated with *msqid* in order to perform this operation.

IPC_MTYP Returns the current **mtype** value and post increments the **mtype** value. The **mtype** value is not allowed to become negative. The **mtype** value is returned in **buf*, which is assumed to be a pointer to a long integer. For further information on use of the **IPC_MTYP** *cmd*, see the discussion of bidirectional queues in "System Calls" of *AIX Operating System Programming Tools and Interfaces.*

Return Value

Upon successful completion, a value of 0 is returned. If the **msgctl** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgctl** system call fails if one or more of the following are true:

EINVAL *msqid* is not a valid message queue identifier.

EINVAL *cmd* is not a valid command.

EACCES *cmd* is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM *cmd* is equal to **IPC_RMID**, **IPC_SET**, or **IPC_RMID2** and the effective user ID of the calling process (translated for a remote queue) is not equal to that of superuser, nor is it equal to the value of **msg_perm.uid** in the data structure associated with *msqid.*

EPERM *cmd* is equal to **IPC_SET**, an attempt is being made to increase to the value of **msg_qbytes**, and the effective user ID of the calling process is not equal to that of superuser.

EFAULT The *buf* parameter points to a location outside of the process's allocated address space.

If Distributed Services is installed on your system, **msgctl** can also fail if one or more of the following are true:

EINVAL The *cmd* specified was **IPC_RMID2**, but *msqid* does not identify a remote queue.

ESTALE The current boot count of the server is the same as when the *msqid* was obtained.

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: “msgget” on page 2-76, “msgrcv” on page 2-79, “msgsnd” on page 2-82, “msgxrcv” on page 2-85,“create_ipc_prof” on page 3-40.2 , “del_ipc_prof” on page 3-64.1, and “find_ipc_prof” on page 3-166.1.

Managing the AIX Operating System.

AIX Operating System Programming Tools and Interfaces.

msgget

Purpose

Gets a message queue identifier.

Syntax

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (*key*, *msgflg***)**
key_t *key*;
int *msgflg*;

Description

The **msgget** system call returns the message queue identifier associated with the specified *key*. The *key* parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See “ftok” on page 3-198 for details about this subroutine.

If Distributed Services is installed on your system, the **msgget** system call provides access to both local and remote queues.

The *msgflg* parameter is constructed by logically OR-ing one or more of the following values:

IPC_CREAT Creates the data structure if it does not already exist.
IPC_EXCL Causes the **msgget** system call to fail if **IPC_CREAT** is also set and the data structure already exists.

S_IRUSR Permits the process that owns the data structure to read it.
S_IWUSR Permits the process that owns the data structure to modify it.
S_IRGRP Permits the group associated with the data structure to read it.
S_IWGRP Permits the group associated with the data structure to modify it.
S_IROTH Permits others to read the data structure.
S_IWOTH Permits others to modify the data structure.

The values that begin with **S_I-** are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

A message queue identifier and associated message queue and data structure are created for the value of the *key* parameter if one of the following are true:

- *key* is equal to **IPC_PRIVATE.**
- *key* does not already have a message queue identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new message queue identifier is initialized as follows:

- **msg_perm.cuid**, **msg_perm.uid**, **msg_perm.cgid**, and **msg_perm.gid** are set equal to the effective user ID and effective group ID, respectively, of the calling process.
- The low-order nine bits of **msg_perm.mode** are set equal to the low-order nine bits of the *msgflg* parameter.
- **msg_qnum**, **msg_lspid**, **msg_lrpid**, **msg_stime**, and **msg_rtime** are set equal to 0.
- **msg_ctime** is set equal to the current time.
- **msg_qbytes** is set equal to the system limit.

The **msgget** system call performs the following actions:

1. First, **msgget** looks in the IPC key mapping tables for *key*.
2. If the tables do not have an entry for *key*, then **msgget** either finds or creates (depending on the value of *msgflg*) a local queue with *key*.
3. If the tables indicate that *key* has been mapped to a new key at a remote node and you have installed Distributed Services, then **msgget**:
   - Allocates a local header for the remote queue.
   - Queries the remote node to find or create a queue with the indicated *key*.
   - Installs the information (handle, boot count, and so on) returned by the remote node into the local header.
4. Finally, **msgget** returns the ID of the local queue header to its caller.

For an explanation of how to set up local to remote key mappings, see the **dstables** command in *AIX Operating System Commands Reference.*

Return Value

Upon successful completion, a message queue identifier is returned. If the **msgget** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgget** system call fails if one or more of the following are true:

EACCES A message queue identifier exists for the *key* parameter but operation permission as specified by the low-order nine bits of the *msgflg* parameter would not be granted.

ENOENT A message queue identifier does not exist for the *key* parameter and **IPC_CREAT** is not set.

ENOSPC A message queue identifier is to be created but the system imposed limit on the maximum number of allowed message queue identifiers system wide would be exceeded.

EEXIST A message queue identifier exists for *key*, and both **IPC_CREAT** and **IPC_EXCL** are set.

If Distributed Services is installed on your system, **msgget** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: “msgctl” on page 2-73, “msgrcv” on page 2-79, “msgsnd” on page 2-82, “msgxrcv” on page 2-85, and “ftok” on page 3-198.

The **dstables** command in *AIX Operating System Commands Reference.*

msgrcv

Purpose

Reads a message from a queue.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv (*msqid*, *msgp*, *msgsz*, *msgtyp*, *msgflg***)**
int *msqid*;
struct msgbuf **msgp*;
int *msgsz*;
long *msgtyp*;
int *msgflg*;

Description

The **msgrcv** system call reads a message from the queue specified by the *msqid* parameter and stores it into the structure pointed to by the *msgp* parameter. The current process must have read permission in order to perform this operation. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;     /* Message type */
char  mtext[1];  /* Beginning of message text */
```

The **mtype** field contains the type of the received message as specified by the sending process. **mtext** is the text of the message. If Distributed Services is installed on your system, messages can be received from either local or remote queues (see "msgsnd" on page 2-82).

The *msgsz* parameter specifies the size of **mtext** in bytes. The received message is truncated to the size specified by the *msgsz* parameter if it is longer than the size specified by the *msgsz* parameter and if **MSG_NOERROR** is set in *msgflg*. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than *msgsz* bytes and **MSG_NOERROR** is not set, then the **msgrcv** system call fails and sets **errno** to **E2BIG**.

The *msgtyp* parameter specifies the type of message requested as follows:

- If the *msgtyp* parameter is equal to 0, the first message on the queue is received.

- If the *msgtyp* parameter is greater than 0, the first message of the type specified by the *msgtyp* parameter is received.
- If the *msgtyp* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *msgtyp* parameter is received.

The *msgflg* parameter is either 0, or is constructed by logically OR-ing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than *msgsz* bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, then the calling process returns a value of -1 and sets **errno** to **ENOMSG**.
- If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier specified by the *msqid* parameter is removed from the system. When this occurs, **errno** is set to **EIDRM**, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner described in "signal" on page 2-145.

Return Value

Upon successful completion, **msgrcv** returns a value equal to the number of bytes actually stored into **mtext** and the following actions are taken with respect to the data structure associated with the *msqid* parameter:

- **msg_qnum** is decremented by 1.
- **msg_lrpid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

If the **msgrcv** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgrcv** system call fails if one or more of the following are true:

EINVAL *msqid* is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL *msgsz* is less than 0.

E2BIG **mtext** is greater than *msgsz* and **MSG_NOERROR** is not set.

ENOMSG The queue does not contain a message of the desired type and **IPC_NOWAIT** is set.

EFAULT The *msgp* parameter points to a location outside of the process's allocated address space.

EINTR **msgrcv** received a signal.

EIDRM The message queue identifier specified by *msqid* has been removed from the system.

If Distributed Services is installed on your system, **msgrcv** can also fail if one or more of the following are true:

ESTALE The current boot count of the server is the same as when the *msqid* was obtained.

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "msgctl" on page 2-73, "msgget" on page 2-76, "msgsnd" on page 2-82, "msgxrcv" on page 2-85, and "signal" on page 2-145.

msgsnd

Purpose

Sends a message.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd (*msqid*, *msgp*, *msgsz*, *msgflg***)**
int *msqid*;
struct msgbuf **msgp*;
int *msgsz*, *msgflg*;

Description

The **msgsnd** system call sends a message to the queue specified by the *msqid* parameter. If Distributed Services is installed on your system, this queue can reside on another node. The current process must have write permission in order to perform this operation. The *msgp* parameter points to a **msgbuf** structure containing the message. The **msgbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
long  mtype;     /* Message type */
char  mtext[1];  /* Beginning of message text */
```

The *mtype* parameter is a positive integer that is used by the receiving process for message selection. The *mtext* parameter is any text of the length in bytes specified by the *msgsz* parameter. The *msgsz* parameter can range from 0 to a system-imposed maximum.

If *msqid* identifies a queue header that is a local key for a remote queue, then the message is sent to the message queue at the remote node.

The *msgflg* parameter specifies the action to be taken if the message cannot be sent for one of the following reasons:

- The number of bytes already on the queue is equal to **msg_qbytes**.
- The total number of messages on all queues system-wide is equal to a system-imposed limit.

These actions are as follows:

- If *msgflg* is set to **IPC_NOWAIT**, then the message is not sent, and **msgsnd** returns a value of -1 and sets **errno** to **EAGAIN**.
- If *msgflg* is 0, then the calling process suspends execution until one of the following occurs:
  - The condition responsible for the suspension no longer exists, in which case the message is sent.
  - *msqid* is removed from the system. (For information on how to remove *msqid*, see "msgctl" on page 2-73.) When this occurs, **errno** is set equal to EIDRM, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case the message is not sent and the calling process resumes execution in the manner prescribed in "signal" on page 2-145.

Return Value

Upon successful completion, a value of 0 is returned and the following actions are taken with respect to the data structure associated with the *msqid* parameter:

- **msg_qnum** is incremented by 1.
- **msg_lspid** is set equal to the process ID of the calling process.
- **msg_stime** is set equal to the current time.

If the **msgsnd** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgsnd** system call fails and no message is sent if one or more of the following are true:

EINVAL The *msqid* parameter is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL **mtype** is less than 1.

EAGAIN The message cannot be sent for one of the reasons stated previously, and *msgflg* is set to **IPC_NOWAIT**.

EINVAL The *msgsz* parameter is less than 0 or greater than the system-imposed limit.

EFAULT The *msgp* parameter points to a location outside of the process's allocated address space.

EINTR **msgsnd** received a signal.

EIDRM The message queue identifier specified by *msqid* has been removed from the system.

If Distributed Services is installed on your system, **msgsnd** can also fail if one or more of the following are true:

ESTALE The current boot count of the server is the same as when the *msqid* was obtained.

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "msgctl" on page 2-73, "msgget" on page 2-76, "msgrcv" on page 2-79, "msgxrcv" on page 2-85, and "signal" on page 2-145.

msgxrcv

Purpose

Receives an extended message.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgxrcv (*msqid*, *msgp*, *msgsz*, *msgtyp*, *msgflg***)**
int *msqid*;
struct msgxbuf **msgp*;
int *msgsz*, *msgflg*;
long *msgtyp*;

Description

The **msgxrcv** system call reads a message from the queue specified by the *msqid* parameter and stores it into the extended message receive buffer pointed to by the *msgp* parameter. The current process must have read permission in order to perform this operation. The **msgxbuf** structure is defined in the **sys/msg.h** header file, and it contains the following members:

```
time_t    mtime;     /* Time and date message was sent */
short     muid;      /* Sender's effective user ID */
short     mgid;      /* Sender's effective group ID */
long      mnid;      /* Sender's node ID */
short     mpid;      /* Sender's process ID */
long      mtype;     /* Message type */
char      mtext[1];  /* Beginning of message text */
```

The *msgsz* parameter specifies the size of **mtext** in bytes. The receive message is truncated to the size specified by the *msgsz* parameter if it is larger than the *msgsz* parameter and **MSG_NOERROR** is true. The truncated part of the message is lost and no indication of the truncation is given to the calling process.

The *msgsz* parameter specifies the size of **mtext** in bytes. The received message is truncated to the size specified by the *msgsz* parameter if it is larger than the size specified by the *msgsz* parameter and if **MSG_NOERROR** is set in *msgflg*. The truncated part of the message is lost and no indication of the truncation is given to the calling process. If the message is longer than *msgsz* bytes and **MSG_NOERROR** is not set, then the **msgrcv** system call fails and sets **errno** to **E2BIG**.

The *msgtyp* parameter specifies the type of message requested as follows:

- If the *msgtyp* parameter is equal to 0, the first message on the queue is received.
- If the *msgtyp* parameter is greater than 0, the first message of the type specified by the *msgtyp* parameter is received.
- If the *msgtyp* parameter is less than 0, the first message of the lowest type that is less than or equal to the absolute value of the *msgtyp* parameter is received.

The *msgflg* parameter is either 0, or is constructed by logically OR-ing one or more of the following values:

MSG_NOERROR Truncates the message if it is longer than *msgsz* bytes.

IPC_NOWAIT Specifies the action to take if a message of the desired type is not on the queue:

- If **IPC_NOWAIT** is set, then the calling process returns a value of -1 and sets **errno** to **ENOMSG**.
- If **IPC_NOWAIT** is not set, then the calling process suspends execution until one of the following occurs:
  - A message of the desired type is placed on the queue.
  - The message queue identifier specified by the *msqid* parameter is removed from the system. When this occurs, **errno** is set to **EIDRM**, and a value of -1 is returned.
  - The calling process receives a signal that is to be caught. In this case, a message is not received and the calling process resumes in the manner prescribed in "signal" on page 2-145.

If Distributed Services is installed on your system, the **msgxrcv** call works with both local and remote queues.

Return Value

Upon successful completion, **msgxrcv** returns a value equal to the number of bytes actually stored into **mtext**, and the following actions are taken with respect to the data structure associated with the *msqid* parameter:

- **msg_qnum** is decremented by 1.
- **msg_lrpid** is set equal to the process ID of the calling process.
- **msg_rtime** is set equal to the current time.

If the **msgxrcv** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **msgxrcv** system call fails if one or more of the following are true:

EINVAL *msqid* is not a valid message queue identifier.

EACCES Operation permission is denied to the calling process.

EINVAL *msgsz* is less than 0.

E2BIG **mtext** is greater than *msgsz* and **MSG_NOERROR** is not set.

ENOMSG The queue does not contain a message of the desired type and **IPC_NOWAIT** is set.

EFAULT The *msgp* parameter points to a location outside of the process's allocated address space.

EINTR **msgxrcv** received a signal.

EIDRM The message queue identifier specified by *msqid* is removed from the system.

If Distributed Services is installed on your system, **msgxrcv** can also fail if one or more of the following are true:

ESTALE The current boot count of the server is the same as when the *msqid* was obtained.

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "msgctl" on page 2-73, "msgget" on page 2-76, and "msgrcv" on page 2-79.

nice

Purpose

Changes the priority of a process.

Syntax

int nice (*incr*)
int *incr*;

Description

The **nice** system call adds the value of the *incr* parameter to the nice value of the calling process. A process's ***nice value*** is a positive number that determines that process's CPU priority. A higher number results in a lower CPU priority.

A maximum nice value of 39 and a minimum nice value of 0 are imposed by the system. If *incr* causes the nice value to fall outside this range, then **nice** sets the nice value to the corresponding limit.

Return Value

Upon successful completion, the new nice value minus 20 is returned. If **nice** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **nice** system call fails and the nice value is not changed if:

EPERM The *incr* parameter is negative or the resulting nice value would be greater than 40, and the effective user ID of the calling process is not superuser.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34.

The **nice** command in *AIX Operating System Commands Reference.*

open

Purpose

Opens a file for reading or writing.

Syntax

#include <fcntl.h>

int open (*path***,** *oflag* [, *mode*]**)**
char **path*;
int *oflag*, *mode*;

Description

The **open** system call opens a file descriptor for the file named by the *path* parameter. If Distributed Services is installed on your system, this path can cross into another node.

Note: Distributed Services does not support remote pipes or special files.

The file status flags are set according to the value of the *oflag* parameter. The *oflag* parameter values are constructed by logically OR-ing flags from the following list:

Note: Do not use **O_RDONLY, O_WRONLY,** or **O_RDWR** together.

O_RDONLY Open for reading only.

O_WRONLY Open for writing only.

O_RDWR Open for reading and writing.

O_NDELAY Open with no delay. This flag may affect subsequent reads and writes.

When opening a FIFO with **O_RDONLY** or **O_WRONLY** set:

- If **O_NDELAY** is set, an **open** for reading-only returns without delay. An **open** for writing-only returns an error if no process currently has the file open for reading.
- If **O_NDELAY** is clear, an **open** for reading-only blocks until a process opens the file for writing. An **open** for writing-only blocks until a process opens the file for reading.

When opening a file associated with a communication line:

- If **O_NDELAY** is set, the **open** returns without waiting for carrier.

- If **O_NDELAY** is clear, the **open** blocks until carrier is present.

When opening a regular file that supports enforced record locks:

- If **O_NDELAY** is set, then reads and writes to portions of the file that are locked by other processes return an error.
- If **O_NDELAY** is clear, then reads and writes to portions of the file that are locked by other processes blocks until the locks are released.

O_APPEND If set, the file pointer is set to the end of the file prior to each write.

O_CREAT If the file exists, this flag has no effect. If the file does not exist, then the file's owner ID is set to the process's effective user ID, the file's group ID is set to the process's effective group ID, and the low-order 12 bits of the file mode are set to the value of the *mode* parameter modified as follows:

- All bits set in the process's file mode creation mask are cleared. (For information about the creation mask, see "umask" on page 2-169.)
- The **S_ISVTX** bit of the mode, which saves the text image after execution, is cleared.

For information about file modes and a list of the mode values, see "chmod" on page 2-18 and "stat.h" on page 5-69.)

O_TRUNC If the file exists, then its length is truncated to 0, and the mode and owner are unchanged. If the file has any outstanding record locks, then **open** fails and the file remains unchanged.

O_EXCL If **O_EXCL** and **O_CREAT** are set, **open** fails if the file exists.

The file pointer used to mark the current position within the file is set to the beginning of the file.

The new file descriptor is set to remain open across **exec** system calls (see "fcntl" on page 2-44).

No process can have more than 200 file descriptors open simultaneously.

Return Value

Upon successful completion, the file descriptor, a nonnegative integer, is returned. If **open** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **open** system call fails, and the named file is not opened if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT **O_CREAT** is not set and the named file does not exist.

EACCES A component of the path prefix denies search permission.

EACCES The type of access specified by the *oflag* parameter is denied for the named file.

EISDIR The named file is a directory and the *oflag* parameter is write or read/write.

EROFS The named file resides on a read-only file system and the *oflag* parameter is write or read/write.

EMFILE Two hundred (200) file descriptors are currently open.

ENXIO The named file is a character special or block special file, and the device associated with this special file does not exist.

ENXIO The named file is a multiplexed special file and either the channel number is outside of the valid range, or no more channels are available.

ENXIO The special file or named pipe resides in a remote node.

ETXTBSY The file is a pure procedure (shared text) file that is being executed and the *oflag* parameter is write or read/write.

EFAULT The *path* parameter points to a location outside of the process's allocated address space.

EEXIST **O_CREAT** and **O_EXCL** are set, and the named file exists.

ENXIO **O_NDELAY** is set, the named file is a FIFO, **O_WRONLY** is set, and no process has the file open for reading.

EAGAIN **O_TRUNC** is set, and the named file contains a record lock owned by another process. See “lockf” on page 2-64 for information about record locks.

EINTR A signal was caught during the **open** system call.

ENFILE The system file table is full.

ENOSPC The directory that would contain the new file cannot be extended.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **open** can also fail if one or more of the following are true:

EINVAL The *path* parameter identifies a remote file that is neither a directory nor a regular file.

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "close" on page 2-25, "creat" on page 2-27, "dup" on page 2-32, "fcntl" on page 2-44, "lockf" on page 2-64, "lseek" on page 2-67, "mknod" on page 2-69, "read, readx" on page 2-106, "umask" on page 2-169, "write, writex" on page 2-184, "stat.h" on page 5-69, and Appendix C, "Writing Device Drivers."

pause

Purpose

Suspends a process until a signal is received.

Syntax

int pause ( )

Description

The **pause** system call suspends the calling process until it receives a signal. The signal must not be one that is ignored by the calling process. **pause** does not affect the action taken upon the receipt of a signal.

If the signal received causes the calling process to terminate, then the **pause** system call does not return.

If the signal is caught by the calling process and control is returned from the signal-catching function, then the calling process resumes execution from the point of suspension; the **pause** system call returns a value of -1 and sets **errno** to **EINTR**. (For information about signal-catching functions, see "signal" on page 2-145.)

Related Information

In this book: "alarm" on page 2-13, "kill" on page 2-60, "signal" on page 2-145, and "wait" on page 2-182.

pipe

Purpose

Creates an interprocess channel.

Syntax

int pipe (*fildes***)**
int *fildes***[2];**

Description

The **pipe** system call creates an interprocess channel called a pipe and returns two file descriptors, *fildes*[**0**] and *fildes*[**1**]. The *fildes*[**0**] file descriptor is opened for reading and *fildes*[**1**] is opened for writing.

A read on file descriptor *fildes*[**0**] accesses the data written to *fildes*[**1**] on a first-in-first-out basis.

When writing, at least 5,120 bytes of data are buffered by the pipe before the writing process is blocked.

Warning: The actions of the **pipe** system call are undefined if the *fildes* parameter points to a location outside of the process's allocated address space.

Return Value

Upon successful completion, a value of 0 is returned. If **pipe** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **pipe** system call fails if one or more the following are true:

EMFILE	199 or more file descriptors are already open.
ENFILE	The system file table is full.

Related Information

In this book: "read, readx" on page 2-106, "select" on page 2-111, and "write, writex" on page 2-184.

The **sh** command in *AIX Operating System Commands Reference.*

plock

Purpose

Locks the process, text, or data in memory.

Syntax

#include <sys/lock.h>

int plock (*op***)**
int *op***;**

Description

The **plock** system call allows the calling process to lock or unlock its text segment (***text lock***), its data segment (***data lock***), or both its text and data segments (***process lock***) into memory. Locked segments are pinned in memory and are immune to all routine paging. The effective user ID of the calling process must be superuser to use this call.

The *op* parameter specifies one of the following operations:

PROCLOCK	Locks text and data segments into memory (process lock).
TXTLOCK	Locks text segment into memory (text lock).
DATLOCK	Locks data segment into memory (data lock).
UNLOCK	Removes locks.

Return Value

Upon successful completion, a value of 0 is returned to the calling process. If **plock** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **plock** system call fails if one or more of the following are true:

EPERM The effective user ID of the calling process is not superuser.

EINVAL The *op* parameter has a value other than **PROCLOCK**, **TXTLOCK**, **DATLOCK**, or **UNLOCK**.

EINVAL *op* is equal to **PROCLOCK** and a process lock, a text lock, or a data lock already exists on the calling process.

EINVAL *op* is equal to **TXTLOCK** and a text lock, or a process lock already exists on the calling process.

EINVAL *op* is equal to **DATLOCK** and data lock, or a process lock already exists on the calling process.

EINVAL *op* is equal to **UNLOCK** and no type of lock exists on the calling process.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, and "fork" on page 2-46.

profil

Purpose

Starts and stops execution profiling.

Syntax

#include <mon.h>

void profil (*shortbuff*, *bufsiz*, *offset*, *scale***)**
— or —
void profil (*profbuff*, **-1, 0, 0)**

short **shortbuff*;
struct prof **profbuff*;
unsigned int *bufsiz*, *offset*, *scale*;

Description

The **profil** system call arranges to record a histogram of periodically sampled values of the calling process's program counter.

If the *bufsiz* parameter has any value but -1, then the parameters to **profil** are interpreted as shown in the first syntax definition. The *shortbuff* parameter points to an area of memory, and its length (in bytes) is given by the *bufsiz* parameter.

After this call, the user's program counter (pc) is examined 60 times a second. The value of the *offset* parameter is subtracted from the pc, and the result is multiplied by the value of the *scale* parameter. If the resulting number is less than *bufsiz* ÷ **sizeof(short)**, then the corresponding **short** inside *shortbuff* is incremented.

The least significant 16 bits of the *scale* parameter are interpreted as an unsigned, fixed-point fraction with a binary point at the left. The most significant 16 bits of *scale* are ignored. For example:

Octal	Hex	Meaning
0177777	0xFFFF	Maps approximately each pair of bytes in the instruction space to a unique **short** in *shortbuff.*
077777	0x7FFF	Maps approximately every four bytes to a **short** in *shortbuff.*
01	0x0001	Maps all instructions to the first **short** in *shortbuff,* producing a noninterrupting core clock.
0	0x0000	Turns profiling off.

Mapping each byte of the instruction space to an individual **short** in *shortbuff* is not possible.

If the second parameter (*bufsize*) has the value -1, then the parameters to **profil** are interpreted as shown in the second syntax definition. In this case, the *offset* and *scale* parameters are ignored, and *profbuff* points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** header file, and it contains the following members:

```
daddr_t  p_low;
daddr_t  p_high;
short_t  *p_buff;
int_t    p_bufsize;
int_t    p_scale;
```

If the **p_scale** member has the value -1, then a value for it is computed based on **p_low**, **p_high**, and **p_bufsize**; otherwise **p_scale** is interpreted like the **scale** argument in the first synopsis. The **p_high** members in successive structures must be in ascending sequence. The array of structures is terminated with a structure containing a **p_high** member set to zero.

Profiling is turned off:

- If the value of the *scale* parameter is 0.
- When an **exec** system call is executed
- If updating the buffer pointed to by the *shortbuff* or *profbuff* parameter would cause a memory fault.

Profiling is rendered ineffective by giving a value of 0 for the *bufsiz* parameter.

Profiling remains on in both the child process and the parent process after a **fork** system call.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fork" on page 2-46, and "monitor" on page 3-248.

The **cc** and **prof** commands in *AIX Operating System Commands Reference.*

ptrace

Purpose

Traces the execution of a child process.

Syntax

```
#include <sys/reg.h>

int ptrace (request, pid, addr, data, buff)
int request, pid, *addr, data, *buff;
```

Description

The **ptrace** system call allows a parent process to control the execution of a child process. **ptrace** is primarily used by utility programs to implement breakpoint debugging. The **sdb** command described in *AIX Operating System Commands Reference* is such a debugging utility.

The child process behaves normally until it encounters a signal, at which time it enters a stopped state and its parent process is notified with the **wait** system call. When the child process is in the stopped state, its parent process can examine and modify its memory image using the **ptrace** system call. Also, the parent process can cause the child process to either terminate or continue, with the possibility of ignoring the signal that caused it to stop.

The *request* parameter determines the action to be taken by the **ptrace** system call and is one of the following:

0 This request must be issued by the child process that is to be traced by its parent. This request sets the child's trace flag that causes the child to be left in a stopped state upon receipt of a signal, rather than the state specified by the *func* parameter of the **signal** system call. The *pid*, *addr*, and *data* parameters are ignored, and a return value is not defined for this request. Do not issue this request if the parent does not expect to trace the child.

Note: The remainder of the requests can only be used by the parent process. For each request, the *pid* parameter is the process ID of the child. The child must be in a stopped state before these requests are made.

1, 2 These requests return the **int** in the child's address space at the location pointed to by the *addr* parameter. Either request **1** or request **2** can be used with with equal results. The *data* parameter is ignored. These requests fail if the value of the *addr* parameter is not in the address space of the child process, in which case a value of -1 is returned, and the parent's **errno** is set to **EIO**.

3 This request returns the **int** from the child's user area of the system's address space that is located at the offset given by the *addr* parameter. (For information about the user area, see the **sys/user.h** header file.) The value of the *addr* parameter must be in the range 0 to **ctob(USIZE)**, and it is rounded down to the the next **int** (word) boundary. (**ctob** and **USIZE** are defined by including the **sys/param.h** header file.) The *data* parameter is ignored. This request fails if the *addr* parameter is outside the user area, in which case a value of -1 is returned to the parent process and the parent process's **errno** is set to **EIO**.

4, 5 These requests write the value of the *data* parameter into the address space of the child process at the **int** pointed to by the *addr* parameter. Either request 4 or request 5 can be used with equal results. Upon successful completion, the value written into the address space of the child process is returned to the parent process. These requests fail if the *addr* parameter points to a location in a pure procedure space and a copy cannot be made. They also fail if the *addr* is out of range. Upon failure, a value of -1 is returned to the parent process and the parent process's **errno** is set to **EIO**.

6 This request writes the value of the *data* parameter into the child's user area of the system's address space at the **int** specified by the *addr* parameter. The value of the *addr* parameter is rounded down to the the next **int** (word) boundary. The following values for *addr* are defined in the **sys/reg.h** header file, and they identify the only entries that can be modified:

R0—R15	General Purpose Registers 0—15
IAR	Instruction Address Register
MQ	Multiply/Quotient Register
CS	Condition Status Register.

7 This request causes the child process to resume execution. If the *data* parameter is 0, all pending signals, including the one that caused the child process to stop, are canceled before the child process resumes execution. If the *data* parameter is a valid signal number, the child process resumes execution as if it had received that signal. Any other pending signals are canceled. The *addr* parameter must be equal to 1 for this request. Upon successful completion, the value of the *data* parameter is returned to the parent process. This request fails if the *data* parameter is not 0 or a valid signal number, in which case a value of -1 is returned to the parent process and the parent process's **errno** is set to **EIO**.

8 This request causes the child process to terminate the same way it would with an **exit** system call.

11 This request returns the contents of one of the general-purpose registers of the child process. The *addr* parameter specifies which of the sixteen 32-bit registers is to be returned. The *data* and *buff* parameters are ignored. This request fails if the value of the *addr* parameter is not between 0 and 15 inclusive. In this case, **ptrace** returns the value -1 and sets the parent's **errno** to **EIO**.

12 This request stores the value of a floating-point register into the location pointed to by the *addr* parameter. The *data* parameter specifies which floating-point register, and it must be a value in the range from 0 to 7, excluding 6. Registers 0 through 5 are eight bytes long, and the status register (register 7) is four bytes long.

14 This request stores the value of the *data* parameter in one of the child process's general-purpose registers. The *addr* parameter specifies the register to be modified. The *buff* parameter is ignored. Upon successful completion, the value of *data* is returned to the parent process. This request fails if the value of the *addr* parameter is not between 0 and 15 inclusive. In this case, **ptrace** returns the value -1 and sets the parent's **errno** to **EIO**.

15 This request sets the floating-point register specified by the *data* parameter to the value pointed to by the *addr* parameter. The *data* parameter must be a value in the range from 0 to 7, excluding 6. Registers 0 through 5 are eight bytes long, and the status register (register 7) is four bytes long.

17 This request reads a block of data from the child process's address space. The *addr* parameter points to the block of data in the child's address space and the *data* parameter gives its length in bytes. The value of the *data* parameter must not be greater than 1024. The *buff* parameter points to the location in the parent's address space into which the data is to be copied. Upon successful completion, **ptrace** returns the value of the *data* parameter. If an error occurs, **ptrace** returns -1 and sets the parent's **errno** to indicate the error. This request fails when one or more of the following are true:

EINVAL The *data* parameter is less than 1 or greater than 1024.

EIO The *addr* parameter is not a valid pointer into the child process's address space.

EFAULT The *buff* parameter does not point to a writable location in the parent process's address space.

19 This request writes a block of data into the child process's address space. The *addr* parameter points to the location in the child's address space to be written into. The *data* parameter gives the length of the block in bytes, and it must not be greater than 1024. The *buff* parameter points to the data in the parent's address space to be copied. Upon successful completion, the value of *data* is returned to the parent. If an error occurs, **ptrace** returns -1 and sets the parent's **errno** to indicate the error.

This request fails when one or more of the following are true:

EINVAL The *data* parameter is less than 1 or greater than 1024.

EIO The *addr* parameter is not a valid pointer into the child process's address space.

EFAULT The *buff* parameter does not point to a readable location in the parent process's address space.

As a security measure, the **ptrace** system call inhibits the set-user-ID facility on subsequent **exec** system calls.

If a traced process initiates an **exec** system call, it stops before executing the first instruction of the new image and shows the signal **SIGTRAP**.

Diagnostics

In general, the **ptrace** system call fails if one or more of the following are true:

EIO The *request* parameter is not one of the values listed.

ESRCH The *pid* parameter identifies a child process that does not exist or has not executed a **ptrace** system call with request 0.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "signal" on page 2-145, and "wait" on page 2-182.

The **sdb** command in *AIX Operating System Commands Reference.*

read, readx

Purpose

Reads from file.

Syntax

int read (*fildes*, *buf*, *nbyte***)**
int *fildes*;
**char ****buf*;
unsigned int *nbyte*;

int readx (*fildes*, *buf*, *nbyte*, *ext***)**
int *fildes*, *ext*;
**char ****buf*;
unsigned int *nbyte*;

Description

The **read** system call reads a set number of bytes into a buffer. The **read** system call reads the number of bytes set by the *nbyte* parameter from the file associated with the *fildes* parameter and places those bytes into the buffer pointed to by the *buf* parameter. If Distributed Services is installed on your system, this file can reside on another node.

The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call.

On devices capable of seeking, the **read** starts at a position in the file given by the file pointer associated with the *fildes* parameter. Upon return from the **read** system call, the file pointer is incremented by the number of bytes actually read.

Devices that are incapable of seeking always read from the current position. The value of a file pointer associated with such a file is undefined.

When attempting to read from an empty pipe (or FIFO):

- If **O_NDELAY** is set, the **read** returns 0.
- If **O_NDELAY** is clear, the **read** blocks until data is written to the file or the file is no longer open for writing.

When attempting to read a file associated with a terminal that has no data currently available:

- If **O_NDELAY** is set, the **read** returns 0.
- If **O_NDELAY** is clear, the **read** blocks until data becomes available.

When attempting to read a regular file that supports enforcement mode record locks, and all or part of the region to be read is currently locked by another process:

- If **O_NDELAY** is set, then the **read** returns -1 and sets **errno** to **EAGAIN**.
- If **O_NDELAY** is clear, then the **read** blocks the calling process until the lock is released.

For more information about record locks, see "lockf" on page 2-64.

If the file has been mapped, the **read** system call reads from a mapped file segment. If the *fildes* file descriptor was used to map the file copy-on-write, then the copy-on-write segment is used. Otherwise, the **read** system call reads from the read-write mapped segment for the file. See "shmat" on page 2-131 for information about mapping files.

The **readx** system call performs the same function as **read**, except that it provides communication with character device drivers that require more information or return more status than **read** can handle.

For files, directories, or special files with drivers that do not handle extended operations, the **readx** system call does exactly what the **read** system call does, and the *ext* parameter is ignored.

Each driver interprets the *ext* parameter in a device-dependent way, either as a value or as a pointer to a communication area. The nonextended **read** system call is equivalent to the extended **readx** system call with an *ext* parameter value of 0. Drivers must apply reasonable defaults when the *ext* parameter value is 0.

Return Value

Upon successful completion, the **read** and **readx** system calls return the number of bytes actually read and placed in the buffer; this number may be less than the value of the *nbyte* parameter if the file is associated with a communication line, or if the number of bytes left in the file is less than the value of the *nbyte* parameter. A value of 0 is returned when an end-of-file has been reached. (For information about communication files, see "ioctl" on page 2-56 and "termio" on page 6-114.) If **read** or **readx** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **read** and **readx** system calls fail if one or more of the following are true:

EBADF *fildes* is not a valid file descriptor open for reading.

EAGAIN An enforcement mode record lock is outstanding in the portion of the file that is to be read.

EFAULT *buf* points to a location outside of the process's allocated address space.

EDEADLK A deadlock would occur if the calling process were to sleep until the region to be read was unlocked.

EINTR A signal was caught during the **read** system call.

If Distributed Services is installed on your system, **read** or **readx** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "creat" on page 2-27, "dup" on page 2-32, "fcntl" on page 2-44, "ioctl" on page 2-56, "lockf" on page 2-64, "open" on page 2-90, "pipe" on page 2-95, "termio" on page 6-114, and Appendix C, "Writing Device Drivers."

reboot

Purpose

Restarts the current virtual machine.

Syntax

int reboot (*dev*)
**char ****dev*;

int reboot ((char *) 0)

int reboot ("VRM"**)**

Description

The **reboot** system call restarts (re-IPLs) the current virtual machine from the block special file specified by the *dev* parameter. If the *dev* parameter is 0, the root device is assumed. The reboot is automatic and brings up **/unix** in the normal, nonmaintenance mode.

If the *dev* parameter is the character string "VRM", then Virtual Resource Manager is also restarted.

The effective user ID of the calling process must be superuser for this call to complete.

Warning: The **reboot** system call does not perform a **sync** operation or write pending output to disk. File systems may be damaged if **reboot** is invoked without first assuring that all disk output has completed.

Return Value

Upon successful completion, the **reboot** system call does not return. If the **reboot** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **reboot** system call fails if one or more of the following are true:

EPERM	The effective user ID of the calling process is not superuser.
ENOENT	The specified special file does not exist.
ENOTBLK	The *dev* parameter does not point to a block device file.

ENXIO	The device associated with the *dev* parameter does not exist or is a remote file.
EFAULT	The *dev* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "iplvm, waitvm" on page 2-58 and "sync" on page 2-163.

The **shutdown** command in *AIX Operating System Commands Reference.*

rename

Purpose

Renames a directory or a file within a filesystem.

Syntax

int rename (*frompath***,** *topath***)**
char ****frompath*, ****topath***;**

Description

The **rename** system call renames a directory or a file within a filesystem. The *frompath* and *topath* parameters must both be either files or directories and must reside on the same node. If Distributed Services is installed on your system, this node can be remote.

For **rename** to execute successfully, the calling process must have write permission to the parent directories of both *frompath* and *topath*, to *frompath*, and to *topath*, if it already exists.

The file or directory named by *frompath* cannot contain the file or directory named by *topath*. If *topath* is an existing file or empty directory, it is replaced by *frompath*. If *topath* is a nonempty directory, **rename** exits with an error.

Return Value

Upon successful completion, the **rename** system call returns a value of 0. If the **rename** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **rename** system call fails and the file or directory name remains unchanged if one or more of the following are true:

ENOTDIR A component of either path prefix is not a directory or *frompath* names a directory and *topath* names a nondirectory.

EISDIR The *topath* parameter names a directory and the *frompath* parameter names a nondirectory.

ENOENT A component of either path does not exist or the file named by *frompath* does not exist.

EACCES Creating the requested link requires writing in a directory with a mode that denies write permission.

EACCES Search permission is denied on a component of either *frompath* or *topath.*

EXDEV The link named by *topath* and the file named by *frompath* are on different file systems.

EROFS The named file resides on a read-only file system.

EFAULT Either *frompath* or *topath* points outside of the process's allocated address space.

EINVAL *frompath* is a parent directory of *topath.*

EEXIST The *topath* parameter is an existing nonempty directory.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **rename** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18 and "mkdir" on page 2-68.1.

The **chmod**, **mkdir**, and **mknod**, and **mvdir** commands in *AIX Operating System Commands Reference.*

rmdir

Purpose

Removes a directory file.

Syntax

rmdir (*path***)**
char ****path*;**

Description

The **rmdir** system call removes the directory specified by the *path* parameter. If Distributed Services is installed on your system, this path can cross into another node. The directory you specify must be empty, and you must have write access to it.

Return Value

Upon successful completion, the **rmdir** system call returns a value of 0. If the **rmdir** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **rmdir** system call fails and the directory is not deleted if one or more of the following are true:

EBUSY The directory is in use as either the mount point for a file system or the current directory of the process that issued the **rmdir**.

EEXIST The directory is not empty.

ENOTDIR A component of the path is not a directory.

ENOENT The named file does not exist.

EACCES A component of the path denies search permission or write permission is denied on the directory containing the link to be removed.

EROFS The named file resides on a read-only file system.

EFAULT *path* points outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **rmdir** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "mkdir" on page 2-68.1, "mknod" on page 2-69, "rename" on page 2-110.1, and "umask" on page 2-169.

select

Purpose

Checks the I/O status of multiple file descriptors and message queues.

Syntax

```
#include <sys/select.h>

int select (nfdsmsgs, readlist, writelist, exceptlist, timeout)
int nfdsmsgs;
struct sellist *readlist, *writelist, *exceptlist;
struct timeval *timeout;
```

Description

The **select** system call checks the specified file descriptors and message queues to see if they are ready for reading (receiving) or writing (sending), or if they have an exceptional condition pending.

Note: The **select** system call applies only to character devices, pipes, and message queues. Not all character device drivers support it. See the descriptions of individual character devices in Chapter 6, "Special Files" for information about whether and how specific device drivers support **select**.

The *nfdsmsgs* parameter specifies the number of file descriptors and the number of message queues to check. The low-order 16 bits give the length of a bit mask that specifies which file descriptors to check; the high-order 16 bits give the size of an array that contains message queue identifiers. If either half of the *nfdsmsgs* parameter is equal to 0, then the corresponding bit mask or array is assumed to not be present.

The *readlist*, *writelist*, and *exceptlist* parameters specify what to check for reading, writing, and exceptions, respectively. Together, they specify the ***selection criteria***. Each of these parameters points to a **sellist** structure, which can specify both file descriptors and message queues. Your program must define the **sellist** structure in the following form:

```
struct sellist
{
    int fdsmask[f];   /* file descriptor bit mask  */
    int msgids[m];    /* message queue identifiers */
};
```

The **fdsmask** array is treated as a bit string in which each bit corresponds to a file descriptor. File descriptor *n* is represented by the bit **(1 < <** *n***)** in the array element **fdsmask[***n* **/ BITS(int)]**. (The **BITS** macro is defined in the **values.h** header file.) Each bit that is set to 1 indicates that the status of the corresponding file descriptor is to be checked. Note that the low-order 16 bits of the *nfdsmsgs* parameter specify the number of *bits* (not elements) in the **fdsmask** array that make up the file descriptor mask. If only part of the last **int** is included in the mask, then the appropriate number of low-order bits are used, and the remaining high-order bits are ignored. If you set the low-order 16 bits of the *nfdsmsgs* parameter to 0, then you must *not* define a **fdsmask** array in the **sellist** structure.

Each **int** of the **msgids** array specifies a message queue identifier whose status is to be checked. Elements with a value of -1 are ignored. The high-order 16 bits of the *nfdsmsgs* parameter specify the number of elements in the **msgids** array. If you set the high-order 16 bits of the *nfdsmsgs* parameter to 0, then you must *not* define a **msgids** array in the **sellist** structure.

If the *timeout* parameter is not a **NULL** pointer, then it points to a structure that specifies the maximum length of time to wait for at least one of the selection criteria to be met. The **timeval** structure is defined in the **sys/select.h** header file, and it contains the following members:

```
int tv_sec;     Seconds
int tv_usec;    Microseconds
```

The number of microseconds specified in *timeout*.**tv_usec**, a value from 0 to 999999, is rounded to the nearest second by the AIX Operating System.

If the *timeout* parameter is a **NULL** pointer, then the **select** system call waits indefinitely, until at least one of the selection criteria is met. If the *timeout* parameter points to a **timeval** structure that contains zeros, then the file and message queue status is polled, and the **select** system call returns immediately.

Note: The arrays specified by *readlist*, *writelist*, and *exceptlist* are the same size because each of these parameters points to the same **sellist** structure type. However, you need not specify the same number of file descriptors or message queues in each. Set the file descriptor bits that are not of interest to 0, and set the extra elements of the **msgids** array to -1.

You can use the **SELLIST** macro defined in the **sys/select.h** header file to define the **sellist** structure. The format of this macro is:

SELLIST(*f*, *m***)** *declarator* . . . ;

where *f* specifies the size of the **fdsmask** array, *m* specifies the size of the *msgids* array, and each *declarator* is the name of a variable to be declared as having this type.

For example, suppose you want to test file descriptors 1, 2, and 35 in addition to five message queues. On the RT PC, which has 32-bit integers, this requires two **ints** for the bit mask. Five **ints** are required to specify the message queue identifiers. The structures can be defined like this:

```
SELLIST(2, 5)  rd, wr, ex;
```

This macro expands to:

```
struct
{
   int  fdsmask[2];
   int  msgids[5];
} rd, wr, ex;
```

Note that the **SELLIST** macro does not define the structure with a tag (that is, as **struct sellist**).

The **SELLIST** macro cannot be used if you specify either half of the *nfdsmsgs* parameter as 0, indicating that one of the arrays is not present. Trying to use `SELLIST(0,5)`, for example, results in a compiler error from defining an array with a dimension of 0. In this case, you must define the structure yourself, including only the desired array.

Return Value

Upon successful completion, the **select** system call returns a value that indicates the total number of file descriptors and message queues that satisfy the selection criteria. The **fdsmask** bit masks are modified so that bits set to 1 indicate file descriptors that meet the criteria. The **msgids** arrays are altered so that message queue identifiers that do not meet the criteria are replaced with a value of -1.

The return value is similar to the *nfdsmsgs* parameter in that the low-order 16 bits give the number of file descriptors, and the high-order 16 bits give the number of message queue identifiers. These values indicate the sum total that meet each of the read, write and exception criteria. Therefore, the same file descriptor or message queue may be counted up to three times.

You can use the **NFDS** and **NMSGS** macros to separate out these two values from the return value. If `rc` contains the value returned from the **select** system call, then `NFDS(rc)` is the number of files selected, and `NMSGS(rc)` is the number of message queues selected.

If the **select** system call fails, then it returns a value of -1 and sets **errno** to indicate the error. In this case, the contents of the structures pointed to by the *readlist*, *writelist*, and *exceptlist* parameters are unpredictable. If the time limit specified by the *timeout* parameter expires, then **select** returns a value of 0.

Diagnostics

The **select** system call fails if one or more of the following is true:

EBADF An invalid file descriptor or message queue identifier is specified.

EINTR A signal was encountered before any of the selected events occurred, or before the time limit expired.

EFAULT The *readlist*, *writelist*, *exceptlist*, or *timeout* parameter points to a location outside of the process's allocated address space.

EINVAL One of the parameters contains an invalid value.

Related Information

In this book: "close" on page 2-25, "fcntl" on page 2-44, "ioctl" on page 2-56, "msgctl" on page 2-73, "msgget" on page 2-76, "msgrcv" on page 2-79, "msgsnd" on page 2-82, "msgxrcv" on page 2-85, "open" on page 2-90, "read, readx" on page 2-106, "write, writex" on page 2-184, "values.h" on page 5-77, Chapter 6, "Special Files," Appendix C, "Writing Device Drivers," and "*dd***select**" on page C-11 .

semctl

Purpose

Controls semaphore operations.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (*semid*, *semnum*, *cmd*, *val***)**
 — or —
int semctl (*semid*, *semnum*, *cmd*, *buf***)**
 — or —
int semctl (*semid*, *semnum*, *cmd*, *array***)**

int *semid***;**
unsigned int *semnum***;**
int *cmd***;**
int *val***;**
struct semid_ds **buf***;**
unsigned short *array*[ ]**;**

Description

The **semctl** system call performs a variety of semaphore control operations as specified by the *cmd* parameter. The data type of the last parameter depends on the value of the *cmd* parameter. It is referred to as *val*, *buf*, or *array* to indicate one of the definitions given in the preceding **Syntax** section.

The first seven *cmd*s get and set the values of a **sem** structure, which is defined in the **sys/sem.h** header file and contains the following members:

```
ushort  semval;   /* Operation permission structure */
short   sempid;   /* ID of last process that did a semop */
ushort  semncnt;  /* No. of processes awaiting semval > cval */
ushort  semzcnt;  /* No. of processes awaiting semval = 0 */
```

The following *cmd*s are executed with respect to the semaphore specified by the *semid* and *semnum* parameters.

GETVAL Returns the value of **semval**, if the current process has read permission.

SETVAL Sets the value of **semval** to the value specified by *val*, if the current process has write permission. When this *cmd* is successfully executed, the **semadj** value corresponding to the specified semaphore is cleared in all processes.

GETPID Returns the value of **sempid**, if the current process has read permission.

GETNCNT Returns the value of **semncnt**, if the current process has read permission.

GETZCNT Returns the value of **semzcnt**, if the current process has read permission.

The following *cmds* return and set every **semval** in the set of semaphores.

GETALL Stores **semvals** into the array pointed to by *array*, if the current process has read permission.

SETALL Sets **semvals** according to the array pointed to by *array*, if the current process has write permission. When this *cmd* is successfully executed, the semadj value corresponding to each specified semaphore is cleared in all processes.

The following *cmds* are also available:

IPC_STAT Stores the current value of each member of the data structure associated with the *semid* parameter into the structure pointed to by *buf*, if the current process has read permission. This structure is defined in **sys/sem.h** and contains the following members:

```
struct ipc_perm sem_perm;  /* Operation permission structure */
struct sem  *sem_base;     /* Pointer to first semaphore in set */
ushort      sem_nsems;     /* Number of semaphores in the set */
ushort      semlcnt;       /* Processes waiting on locked semaphore */
time_t      sem_otime;     /* Time of last semop call */
time_t      sem_ctime;     /* Time of the last change to this */
                           /*  structure with a semctl call */
```

IPC_SET Sets the value of the following members of the data structure associated with the *semid* parameter to the corresponding value found in the structure pointed to by *buf*:

```
sem_perm.uid
sem_perm.gid
sem_perm.mode   /* Only the low-order nine bits */
```

This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with the *semid* parameter.

IPC_RMID Removes the semaphore identifier specified by the *semid* parameter from the system and destroys the set of semaphores and data structures associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **sem_perm.uid** in the data structure associated with the *semid* parameter.

Return Value

Upon successful completion, the value returned depends on the *cmd* parameter as follows:

cmd	**Return Value**
GETVAL	Returns the value of **semval**.
GETPID	Returns the value of **sempid**.
GETNCNT	Returns the value of **semncnt**.
GETZCNT	Returns the value of **semzcnt**.
All others	Return a value of 0.

If **semctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **semctl** system call fails if one or more of the following are true:

EINVAL The *semid* parameter is not a valid semaphore identifier.

EINVAL The *semnum* parameter is less than 0 or greater than **sem_nsems**.

EINVAL The *cmd* parameter is not a valid command.

EACCES Operation permission is denied to the calling process.

ERANGE The *cmd* parameter is **SETVAL** or **SETALL** and the value to which **semval** is to be set is greater than the system-imposed maximum.

EPERM The *cmd* parameter is equal to **IPC_RMID** or **IPC_SET** and the effective user ID of the calling process is not equal either to that of superuser or to the value of **sem_perm.uid** in the data structure associated with the *semid* parameter.

EFAULT The *buf* or *array* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "semget" on page 2-119 and "semop" on page 2-122.

semget

Purpose

Gets a set of semaphores.

Syntax

```
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key, nsems, semflg)
key_t key;
int nsems, semflg;
```

Description

The **semget** system call returns the semaphore identifier associated with the specified *key*. The *key* parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See "ftok" on page 3-198 for details about this subroutine. The *nsems* parameter specifies the number of semaphores in the set.

The *semflg* parameter is constructed by logically OR-ing one or more of the following values:

IPC_CREAT Creates the data structure if it does not already exist.
IPC_EXCL Causes the **semget** system call to fail if **IPC_CREAT** is also set and the data structure already exists.

S_IRUSR Permits the process that owns the data structure to read it.
S_IWUSR Permits the process that owns the data structure to modify it.
S_IRGRP Permits the group associated with the data structure to read it.
S_IWGRP Permits the group associated with the data structure to modify it.
S_IROTH Permits others to read the data structure.
S_IWOTH Permits others to modify the data structure.

The values that begin with **S_I-** are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

The **semget** system call creates a data structure for the semaphore ID and an array containing *nsems* semaphores if one of the following is true:

- The *key* parameter is equal to **IPC_PRIVATE**.
- The *key* parameter does not already have a semaphore identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new semaphore identifier is initialized as follows:

- **sem_perm.cuid** and **sem_perm.uid** are set equal to the effective user ID of the calling process.
- **sem_perm.cgid** and **sem_perm.gid** are set equal to the effective group ID of the calling process.
- The low-order nine bits of **sem_perm.mode** are set equal to the low-order nine bits of the *semflg* parameter.
- **sem_nsems** is set equal to the value of the *nsems* parameter.
- **sem_otime** is set equal to 0 and **sem_ctime** is set equal to the current time.

If the *key* parameter is not **IPC_PRIVATE**, **IPC_EXCL** is not set, and a semaphore identifier already exists for the specified *key*, then the value of the *nsems* parameter specifies the number of semaphores that the current process needs. If the *nsems* parameter is 0, then any number of semaphores is acceptable. If the *nsems* parameter is not 0, then the **semget** system call fails if the set contains fewer than *nsems* semaphores.

Return Value

Upon successful completion, a semaphore identifier is returned. If **semget** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **semget** system call fails if one or more of the following are true:

EINVAL The *nsems* parameter is less than 0, equal to 0, or greater than the system-imposed limit.

EACCES A semaphore identifier exists for the *key* parameter but operation permission, as specified by the low-order nine bits of the *semflg* parameter, is not granted.

EINVAL A semaphore identifier exists for the *key* parameter, but the number of semaphores in the set associated with it is less than the value of the *nsems* parameter and the *nsems* parameter is not equal to 0.

ENOENT A semaphore identifier does not exist for the *key* parameter and **IPC_CREAT** is not set.

ENOSPC A semaphore identifier is to be created, but doing so would exceed the maximum number of identifiers allowed system wide.

EEXIST A semaphore identifier exists for the *key* parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

Related Information

In this book: "semctl" on page 2-115, "semop" on page 2-122, and "ftok" on page 3-198.

semop

Purpose

Performs semaphore operations.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (*semid*, *sops*, *nsops***)**
int *semid*;
struct sembuf *sops*[ ];
unsigned int *nsops*;

Description

The **semop** system call performs operations on the set of semaphores associated with the semaphore identifier specified by the *semid* parameter. The *sops* parameter points to an array of structures, each of which specifies a semaphore operation. The *nsops* parameter is the number of such structures in the array. The **sembuf** structure is defined in the **sys/sem.h** header file, and it contains the following members:

```
ushort  sem_num;  /* Semaphore number */
short   sem_op;   /* Semaphore operation */
short   sem_flg;  /* Operation flags */
```

Each semaphore operation specified by a **sem_op** is performed on the corresponding semaphore specified by *semid* and **sem_num**. The **sem_flg** for each operation is either 0, or is constructed by logically OR-ing one or more of the following values:

SEM_UNDO Specifies whether to modify **semadj** values.

SEM_ORDER Specifies whether to perform the operations atomically or individually. (This applies only to the **sem_flg** of the first operation specified in the *sops* array.)

IPC_NOWAIT Specifies whether to wait or to return immediately when a semaphore's **semval** is not a certain value.

If **SEM_ORDER** is not set in *sops*[**0**]**.sem_flg** (the default), then all of the semaphore operations specified in the *sops* array are performed atomically. This means that no

semval value for any **sem_num** that appears in the entire array of operations is modified until all the semaphore operations can be completed. If the calling process must wait until some **semval** requirement is met, then the **semop** system call does so before performing any of the operations. If any semaphore operation would cause an error to occur, then none of the operations are performed.

If **SEM_ORDER** is set in *sops*[**0**]**.sem_flg**, then the operations are performed individually in the order that they appear in the *sops* array, regardless of whether any of the operations require the process to wait. If an operation encounters an error condition, then the **semop** system call sets **SEM_ERR** in the **sem_flg** of the failing operation, sets **errno** to indicate the error, and returns a value of -1. In this case, the operations that precede the failing one in the *sops* array have been performed, but those following it have not.

The action taken for **SEM_UNDO** and **IPC_NOWAIT** is described in the following text.

The **sem_op** field of the **sembuf** structure specifies one of the following three semaphore operations:

1. If **sem_op** is a positive integer and the current process has write permission, then the value of **sem_op** is added to **semval.** If **SEM_UNDO** is set in **sem_flg**, then the value of **sem_op** is also subtracted from the calling process's **semadj** value for the specified semaphore.

2. If **sem_op** is a negative integer and the current process has write permission, then one of the following occurs:

   - If **semval** is greater than or equal to the absolute value of **sem_op**, the absolute value of **sem_op** is subtracted from **semval.** Also, if **SEM_UNDO** is set in **sem_flg**, the absolute value of **sem_op** is added to the calling process's **semadj** value for the specified semaphore. The **exit** system call adds the **semadj** value to the semaphore's **semval** when the process terminates (see "exit, _exit" on page 2-40).

   - If **semval** is less than the absolute value of **sem_op** and **IPC_NOWAIT** is set in **sem_flg**, **semop** returns a value of -1 and sets **errno** to **EAGAIN.**

   - If **semval** is less than the absolute value of **sem_op** and **IPC_NOWAIT** is not set in **sem_flg**, then **semop** increments the **semncnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:

     - **semval** becomes greater than or equal to the absolute value of **sem_op**. When this occurs, the value of semncnt associated with the specified semaphore is decremented, the absolute value of **sem_op** is subtracted from **semval** and, if **SEM_UNDO** is set in **sem_flg**, the absolute value of **sem_op** is added to the calling process's **semadj** value for the specified semaphore.

     - The *semid* for which the calling process is awaiting action is removed from the system (see "semctl" on page 2-115). When this occurs, **errno** is set equal to **EIDRM**, and a value of -1 is returned.

  - The calling process receives a signal that is to be caught. When this occurs, the value of semncnt associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the **signal** system call.

3. If **sem_op** is 0 and the current process has read permission, then one of the following occurs:
   - If **semval** is 0, then **semop** returns a value of 0.
   - If **semval** is not equal to 0 and **IPC_NOWAIT** is set in **sem_flg**, then **semop** returns a value of -1 and sets **errno** to **EAGAIN.**
   - If **semval** is not equal to 0 and **IPC_NOWAIT** is not set in **sem_flg, semop** increments the **semzcnt** associated with the specified semaphore and suspends execution of the calling process until one of the following occurs:
     - **semval** becomes 0, at which time the value of **semzcnt** associated with the specified semaphore is decremented.
     - The *semid* for which the calling process is awaiting action is removed from the system. When this occurs, **errno** is set equal to **EIDRM**, and a value of of -1 is returned.
     - The calling process receives a signal that is to be caught. When this occurs, the value of **semzcnt** associated with the specified semaphore is decremented, and the calling process resumes execution in the manner prescribed in the **signal** system call.

Return Value

Upon successful completion, the **semop** system call returns a value of 0. Also, the **sempid** value for each semaphore that is operated upon is set to the process ID of the calling process.

If **semop** fails, a value of -1 is returned and **errno** is set to indicate the error. If **SEM_ORDER** was set in the **sem_flg** for the first semaphore operation in the *sops* array, then **SEM_ERR** is set in the **sem_flg** for the failing operation.

Diagnostics

The **semop** system call fails if one or more of the following are true for any of the semaphore operations specified by the *sops* parameter. If the operations were performed individually, then see the preceding discussion of **SEM_ORDER** for more information about error situations.

EINVAL The *semid* parameter is not a valid semaphore identifier.

EFBIG **sem_num** is less than 0 or it is greater than or equal to the number of semaphores in the set associated with the *semid* parameter.

E2BIG The *nsops* parameter is greater than the system-imposed maximum.

EACCES Operation permission is denied to the calling process.

EAGAIN The operation would result in suspension of the calling process, but **IPC_NOWAIT** is set in **sem_flg**

ENOSPC The limit on the number of individual processes requesting a **SEM_UNDO** would be exceeded.

EINVAL The number of individual semaphores for which the calling process requests a **SEM_UNDO** would exceed the limit.

ERANGE An operation would cause a **semval** to overflow the system-imposed limit.

ERANGE An operation would cause a **semadj** value to overflow the system-imposed limit.

EFAULT The *sops* parameter points to a location outside of the process's allocated address space.

EINTR The **semop** system call received a signal.

EIDRM The semaphore identifier *semid* has been removed from the system.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, "fork" on page 2-46, "semctl" on page 2-115, and "semget" on page 2-119.

setgroups

Purpose

Sets the group access list.

Syntax

#include <grp.h>

int setgroups (*ngroups*, *gidset)***;**
int *ngroups*, ******gidset*;

Description

The **setgroups** system call sets the group access list of the current user process according to the array pointed to by the *gidset* parameter. The *ngroups* parameter indicates the number of entries in the array and must not be more than **NGROUPS**, as defined in the **grp.h** header file. Only a process with an effective user ID of superuser can set new groups.

Return Value

Upon successful completion, a value of 0 is returned. If the **setgroups** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **setgroups** system call fails if one or more of the following is true:

EPERM The caller is not superuser.

EINVAL The value of the *ngroups* parameter is greater than **NGROUPS**.

EFAULT The *gidset* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "getgroups" on page 2-52 and "initgroups" on page 3-230.

setpgrp

Purpose

Sets the process group ID.

Syntax

int setpgrp (*flag***)**
int *flag*;

Description

If the *flag* parameter has a nonzero value, then **setpgrp** sets the process group ID of the calling process to be the same as its process ID and returns the new value. If the *flag* parameter is 0, then the process group ID is not changed, but its value is returned.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fork" on page 2-46, "getpid, getpgrp, getppid" on page 2-54, "kill" on page 2-60, and "signal" on page 2-145.

setuid, setgid

Purpose

Sets a process's user and groups IDs.

Syntax

int setuid (*uid*)
int *uid*;

int setgid (*gid*)
int *gid*;

Description

The **setuid** system call sets the real and effective user IDs of the calling process. If the effective user ID of the calling process is superuser, then the real and effective user IDs are set to the value of the *uid* parameter. If the effective user ID of the calling process is not superuser, but the real user ID is equal to the value of the *uid* parameter, or the process's original effective user ID as set by the **exec** system call is equal to *uid*, then the effective user ID is set to the value of the *uid* parameter.

The **setgid** system call sets the real and effective group IDs of the calling process. If the effective user ID of the calling process is superuser, then the real and effective group IDs are set to the value of the *gid* parameter. If the effective user ID of the calling process is not superuser, but the real group ID is equal to the value of the *gid* parameter, or the process's original effective group ID as set by the **exec** system call is equal to *gid*, then the effective group ID is set to the value of the *gid* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If the **setuid** or **setgid** system call fails, then a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **setuid** and **setgid** system calls fail if the following is true:

EPERM The *uid* (*gid*) parameter is not equal to the real user (group) ID of the process or to the original effective user (group) ID as set by the **exec** system call, and the effective user ID is not superuser.

EINVAL The *uid* parameter is not a valid user ID.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "getpid, getpgrp, getppid" on page 2-54, and "getuid, geteuid, getgid, getegid" on page 2-55.

shmat

Purpose

Attaches a shared memory segment or a mapped file to the current process.

Syntax

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;
```

Description

The **shmat** system call attaches the shared memory segment or mapped file associated with the shared memory identifier (returned by **shmget**) or file descriptor (returned by **open**) specified by the *shmid* parameter to the address space of the calling process.

Note: You cannot map a remote file.

The segment or file is attached at the address specified by the *shmaddr* parameter as follows:

- If the *shmaddr* parameter is equal to 0, the segment or file is attached at the first available address as selected by the system.
- If the *shmaddr* parameter is **not** equal to 0, and **SHM_RND** is set in *shmflg*, the segment or file is attached at the next lower segment boundary. This address given by (*shmaddr* - (*shmaddr* modulo **SHMLBA**)).
- If the *shmaddr* parameter is *not* equal to 0 and **SHM_RND** *not* set in *shmflg*, the segment or file is attached at the address given by the *shmaddr* parameter. If this address does not point to a segment boundary, then the **shmat** system call returns the value -1 and sets **errno** to **EINVAL**.

The *shmflg* parameter specifies several options. Its value is either 0, or is constructed by logically OR-ing one or more of the following values:

SHM_RND Rounds the address given by the *shmaddr* parameter to the next lower segment boundary, if necessary.

SHM_MAP Maps a file onto the address space instead of a shared memory segment. The *shmid* must specify an open file descriptor in this case.

SHM_RDONLY Specifies read-only mode instead of the default read-write mode.

SHM_COPY Maps a file in copy-on-write mode.

Either **SHM_RDONLY** or **SHM_COPY** may be specified, but not both.

If **SHM_MAP** is not set in *shmflg*, then a shared memory segment is attached to the data segment. It is attached for reading if **SHM_RDONLY** is set in *shmflg* and if the current process has read permission. If **SHM_RDONLY** is not set and the current process has both read and write permission, then it is attached for reading and writing.

If **SHM_MAP** is set in *shmflg*, then file mapping takes place. In this case, the **shmat** system call maps the file open on file descriptor *shmid* onto a segment. The file must be a regular file. The segment is then mapped into the process's address space.

When file mapping is requested, the *shmflg* parameter specifies how the file is to be mapped. If **SHM_RDONLY** is set, then the file is mapped read-only. If **SHM_COPY** is set, then the file is mapped copy-on-write. If neither of these cases is true, then the file is mapped read-write. The file must be opened for writing before it can be mapped read-write or copy-on-write.

All processes that map the same file read-only or read-write map to the same segment. This segment remains mapped until the last process mapping the file closes it.

All processes that map the same file copy-on-write map the same copy-on-write segment. Changes to the shared segment do not affect the contents of the file resident in the file system until an **fsync** system call is issued for a file descriptor for which copy-on-write mapping was requested. If a process requests copy-on-write mapping for a file and the copy-on-write segment does not yet exist, then it is created, and that segment is maintained for sharing until the last process attached to it detaches it with a **close** system call. When the mapped file is closed, the segment is detached. The next request for copy-on-write mapping for the same file causes a new segment to be created for the file.

A file descriptor can be used to map the corresponding file only once. A file can be multiply mapped by using multiple file descriptors. However, a file cannot be mapped both read-write and copy-on-write by one or more users at the same time. The results are unpredictable if a file that one process has mapped copy-on-write is modified by another process with the **write** system call, unless that process has also attached the copy-on-write segment with a **shmat** system call.

When a file is mapped onto a segment, the file is referenced by accessing the segment. The memory paging system automatically takes care of the physical I/O. References beyond the end of the file cause the file to be extended in increments of the page size.

Return Value

Upon successful completion, the segment start address of the attached shared memory segment or mapped file is returned. If **shmat** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmat** system call fails and the shared memory segment or mapped file is not attached if one or more of the following are true:

EACCES	Operation permission is denied to the calling process.
ENOMEM	The available data space in memory is not large enough to hold the shared memory segment.
ENOMEM	The available data space in memory is not large enough to hold the mapped file data structure.
EINVAL	The *shmid* parameter is not a valid shared memory identifier, or the file to be mapped resides in a remote node.
EINVAL	The *shmaddr* parameter is not equal to 0, and the value of (*shmaddr* - (*shmaddr* modulo **SHMLBA**)) points to a location outside of the process's allocated address space.
EINVAL	The *shmaddr* parameter is not equal to 0, **SHM_RND** is not set in *shmflg*, and the the *shmaddr* parameter points to a location outside of the process's allocated address space.
EINVAL	The *shmaddr* parameter is not equal to 0, **SHM_RND** is not set in *shmflg*, and the the *shmaddr* parameter does not point to a segment boundary.
EEXIST	The file to be mapped has already been mapped.
ETXTBSY	The **shmat** system call attempted to map a file onto a segment attached to a shared library.
EMFILE	The number of shared memory segments attached to the calling process would exceed the system-imposed limit.
EBADF	A file descriptor to map does not refer to an open regular file, or both read-only and copy-on-write modes were requested.
EACCES	A file to be mapped is open read-only, but the segment is to be mapped read-write or copy-on-write.
EACCES	The file is to be mapped read-write, but the file is currently mapped copy-on-write; or the file is to be mapped copy-on-write, but it is currently mapped read-write.

EACCES The file to be mapped has enforced locking enabled, and the file is currently locked.

EFBIG The file to be mapped is larger than the maximum size of a segment.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, "fclear" on page 2-42, "fork" on page 2-46, "fsync" on page 2-48, "ftruncate" on page 2-50, "read, readx" on page 2-106, "shmctl" on page 2-135, "shmdt" on page 2-138, "shmget" on page 2-140, and "write, writex" on page 2-184.

shmctl

Purpose

Controls shared memory operations.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (*shmid*, *cmd*, *buf***)**
int *shmid*, *cmd*;
struct shmid_ds **buf*;

Description

The **shmctl** system call performs a variety of shared memory control operations as specified by the *cmd* parameter. The *shmid* parameter is a shared memory identifier returned by the **shmget** system call. The following *cmds* are available:

IPC_STAT Places the current value of each member of the data structure associated with the *shmid* parameter into the **shmid_ds** structure pointed to by the *buf* parameter. The current process must have read permission in order to perform this operation. The **shmid_ds** structure is defined in the **sys/shm.h** header file, and it contains the following members:

```
struct ipc_perm shm_perm;   /* Operation permission structure */
int     shm_segsz;          /* Segment size */
ushort  shm_segid;          /* Segment identifier */
ushort  shm_lpid;           /* ID of last process to call shmop */
ushort  shm_cpid;           /* ID of process that created this shmid */
ushort  shm_nattch;         /* Current number of processes attached */
ushort  shm_cnattach;       /* No. of in-memory processes attached */
time_t  shm_atime;          /* Time of last shmat call */
time_t  shm_dtime;          /* Time of last shmdt call */
time_t  shm_ctime;          /* Time of the last change to this */
                            /*  structure with a shmctl call */
```

IPC_SET Sets the value of the following members of the data structure associated with the *shmid* parameter to the corresponding value found in the structure pointed to by the *buf* parameter:

```
shm_perm.uid
shm_perm.gid
shm_perm.mode  /* Only the low-order nine bits */
```

This *cmd* can only be performed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the *shmid* parameter.

IPC_RMID Removes the shared memory identifier specified by the *shmid* parameter from the system and erases the shared memory segment and data structure associated with it. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the *shmid* parameter.

SHM_SIZE Sets the size of the shared memory segment to the value specified by *buf*->**shm_segsz**. This value can be larger or smaller than the current size, as long as it is not greater than the value of the **shmmax** keyword set in the **/etc/master** file. This *cmd* can only be executed by a process that has an effective user ID equal to either that of superuser or to the value of **shm_perm.uid** in the data structure associated with the *shmid* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If **shmctl** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmctl** system call fails if one or more of the following are true:

EINVAL The *shmid* parameter is not a valid shared memory identifier.

EINVAL The *cmd* parameter is not a valid command.

EINVAL The *cmd* parameter is equal to **SHM_SIZE** and *buf*->**shm_segsz** is greater than the value of the **shmmax** keyword in the **/etc/master** file.

EACCES The *cmd* parameter is equal to **IPC_STAT** and read permission is denied to the calling process.

EPERM The *cmd* parameter is equal to **IPC_RMID**, **IPC_SET**, or **SHM_SIZE**, and the effective user ID of the calling process is neither equal to the superuser ID, nor is it equal to the value of **shm_perm.uid** in the data structure associated with *shmid*.

ENOMEM The *cmd* parameter is equal to **SHM_SIZE** and the attempt to change the segment size failed.

EFAULT The *buf* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "disclaim" on page 2-30, "shmat" on page 2-131, "shmdt" on page 2-138, "shmget" on page 2-140, and "master" on page 4-98.

shmdt

Purpose

Detaches a shared memory segment.

Syntax

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt (*shmaddr*)
char *_shmaddr_;

Description

The **shmdt** system call detaches, from the calling process's data segment, the shared memory segment located at the address specified by the *shmaddr* parameter.

Mapped file segments are automatically detached when no longer in use. However, you can use the **shmdt** system call to explicitly release the segment register used to map a file. Shared memory segments must be explicitly detatched with **shmdt**.

Return Value

Upon successful completion, a value of 0 is returned. If **shmdt** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmdt** system call fails and the shared memory segment is not detached if the following is true:

EINVAL The *shmaddr* parameter is not the data segment start address of a shared memory segment.

ETXTBSY The **shmdt** system call attempted to detach a segment attached to a shared library.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, "fclear" on page 2-42, "fork" on page 2-46, "fsync" on page 2-48, "shmat" on page 2-131, "shmctl" on page 2-135, and "shmget" on page 2-140.

shmget

Purpose

Gets shared memory segment.

Syntax

#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (*key*, *size*, *shmflg***)**
key_t *key***;**
int *size*, *shmflg***;**

Description

The **shmget** system call returns the shared memory identifier associated with the specified *key*. The *key* parameter is either the value **IPC_PRIVATE** or an IPC key constructed by the **ftok** subroutine (or by a similar algorithm). See "ftok" on page 3-198 for details about this subroutine. The *size* parameter specifies the number of bytes of shared memory required.

The *shmflg* parameter is constructed by logically OR-ing one or more of the following values:

IPC_CREAT Creates the data structure if it does not already exist.
IPC_EXCL Causes the **shmget** system call to fail if **IPC_CREAT** is also set and the data structure already exists.

S_IRUSR Permits the process that owns the data structure to read it.
S_IWUSR Permits the process that owns the data structure to modify it.
S_IRGRP Permits the group associated with the data structure to read it.
S_IWGRP Permits the group associated with the data structure to modify it.
S_IROTH Permits others to read the data structure.
S_IWOTH Permits others to modify the data structure.

The values that begin with **S_I-** are defined in the **sys/stat.h** header file and are a subset of the access permissions that apply to files.

A shared memory identifier, its associated data structure, and a shared memory segment equal in bytes to the value of the *size* parameter are created for the *key* parameter if one of the following is true:

- The *key* parameter is equal to **IPC_PRIVATE**.
- The *key* parameter does not already have a shared memory identifier associated with it, and **IPC_CREAT** is set.

Upon creation, the data structure associated with the new shared memory identifier is initialized as follows:

- **shm_perm.cuid** and **shm_perm.uid** are set equal to the effective user ID of the calling process.
- **shm_perm.cgid** and **shm_perm.gid** are set equal to the effective group ID of the calling process.
- The low-order nine bits of **shm_perm.mode** are set equal to the low-order nine bits of the *shmflg* parameter.
- **shm_segsz** is set equal to the value of the *size* parameter.
- **shm_lpid**, **shm_nattch**, **shm_atime**, and **shm_dtime** are set equal to 0.
- **shm_ctime** is set equal to the current time.

Return Value

Upon successful completion, a shared memory identifier is returned. If **shmget** fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **shmget** system call fails if one or more of the following are true:

EINVAL The *size* parameter is less than the system-imposed minimum or greater than the system-imposed maximum.

EACCES A shared memory identifier exists for the *key* parameter but operation permission as specified by the low-order nine bits of the *shmflg* parameter is not granted.

EINVAL A shared memory identifier exists for *key*, but the size of the segment associated with it is less than the *size* parameter and the *size* parameter is not equal to 0.

ENOENT A shared memory identifier does not exist for the *key* parameter and **IPC_CREAT** not set.

ENOSPC A shared memory identifier is to be created but the system-imposed limit on the maximum number of allowed shared memory identifiers system wide will be exceeded.

ENOMEM A shared memory identifier and associated shared memory segment are to be created but the amount of available physical memory is not sufficient to fill the request.

EEXIST A shared memory identifier exists for the *key* parameter, and both **IPC_CREAT** and **IPC_EXCL** are set.

Related Information

In this book: "shmat" on page 2-131, "shmctl" on page 2-135, "shmdt" on page 2-138, and "ftok" on page 3-198.

sigblock

Purpose

Blocks signals.

Syntax

int sigblock (*mask*)
int *mask*;

Description

The **sigblock** system call causes the signals specified by the *mask* parameter to be added to the set of signals currently being blocked from delivery. The signals are blocked from delivery by logically OR-ing the *mask* parameter into the process's signal mask. Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

It is not possible to block **SIGKILL**. The system provides no indication of this restriction.

Typically, the **sigblock** system call is used to block signals during a critical section of code, and then **sigsetmask** is called to restore the mask to the previous value returned by **sigblock**.

Return Value

Upon completion, the previous set of masked signals is returned.

Example

The following example sets the signal mask to block **SIGINT** from delivery, in addition to the signals that are already blocked:

```
#include <signal.h>

int prevmask;
 . . .
prevmask = sigblock (1 << (SIGINT - 1));
```

Related Information

In this book: “kill” on page 2-60, “signal” on page 2-145, “sigvec” on page 2-156, and “sigsetmask” on page 2-152.

signal

Purpose

Specifies the action to take upon receipt of a signal.

Syntax

```
#include <sys/signal.h>

int (*signal (sig, action)) ( )
int sig;
void (*action) ( );
```

Description

The **signal** system call allows the calling process to choose one of three ways to handle the receipt of a specific signal. The *sig* parameter specifies the signal and the *action* parameter specifies the choice.

The *sig* parameter can be any one of the following signal values except **SIGKILL**. Each of the names shown below is defined in the **sys/signal.h** header file with the value of the corresponding signal number.

SIGHUP	1	Hangup
SIGINT	2	Interrupt
SIGQUIT	3*	Quit
SIGILL	4*	Illegal instruction (not reset when caught)
SIGTRAP	5*	Trace trap (not reset when caught)
SIGIOT	6*	Abort process (see "abort" on page 3-5)
SIGDANGER	7+	The system is likely to "crash" soon
SIGFPE	8*+	Arithmetic exception, integer divide by 0, or floating point exception
SIGKILL	9	Kill (cannot be caught or ignored)
SIGBUS	10*	Specification exception
SIGSEGV	11*	Segmentation violation
SIGSYS	12*	Bad parameter to system call
SIGPIPE	13	Write on a pipe when there is no process to read it

SIGALRM	14	Alarm clock
SIGTERM	15	Software termination signal
SIGUSR1	16	User-defined signal 1
SIGUSR2	17	User-defined signal 2
SIGCLD	18+	Death of a child process
SIGPWR	19+	Power-fail restart (not reset when caught)
SIGAIO	25	Basic LAN signal for asynchronous I/O
SIGPTY	26	PTY device driver read/write availability
SIGIOINT	27	I/O intervention required
SIGGRANT	28#	HFT monitor access wanted
SIGRETRACT	29#	HFT monitor access should be relinquished
SIGSOUND	30#	An HFT sound control has completed execution
SIGMSG	31#	Input data has been stored into the HFT monitor mode ring buffer

The symbols in the preceding table have the following meaning:

* A memory image file (**core** file) is created when one of these signals is received. This is explained in more detail in the following discussion of **SIG_DFL.**

+ These signals require special consideration, as described in "Special Signals" on page 2-148.

For more information on the use of these signals, see "hft" on page 6-23.

The *action* parameter is one of three values: **SIG_DFL**, **SIG_IGN**, or a *function address.* The actions prescribed by these values of are as follows:

SIG_DFL — Default action: Terminate process upon receipt of signal.

Upon receipt of the signal *sig*, the receiving process is to be terminated with all of the consequences outlined in the **exit** system call. In addition, a *memory image* file will be created in the current directory of the receiving process if *sig* is one for which an asterisk appears in the preceding list *and* the following conditions are met:

- The effective user ID and the real user ID of the receiving process are equal.
- An ordinary file named **core** exists in the current directory and is writable, or it can be created. If the file must be created, it will have the following properties:
  - The access permission code 0666 (0x1B6), modified by the file creation mask (see "umask" on page 2-169)
  - A file owner ID that is the same as the effective user ID of the receiving process
  - A file group ID that is the same as the effective group ID of the receiving process.

SIG_IGN — Ignore signal.

The signal *sig* is to be ignored.

Note: The **SIGKILL** signal cannot be ignored.

function address — Catch signal.

Upon receipt of the signal *sig*, the receiving process is to execute the signal-catching function pointed to by the *action* parameter. The signal number *sig* is passed as the only parameter to the signal-catching function. Before calling the signal-catching function, the value of *action* for the caught signal is set to **SIG_DFL** unless the signal is **SIGILL, SIGTRAP**, or **SIGPWR**.

When the signal-catching function returns, the value of the signal mask upon entry is restored, and the receiving process resumes execution at the point at which it was interrupted.

Note that after a signal is received, there is a period of time during which the signal action is set to **SIG_DFL** and the signal-catching function has not had a chance to re-establish itself as the catcher for this signal. If the signal occurs again during that period, it will not be caught. The **sigvec** system call offers an enhanced signal-handling capacity to avoid this ***race condition***.

When a signal that is to be caught occurs during a **read**, **write**, **open**, or **ioctl** system call on a slow device (like a terminal; but not an ordinary file), during a **pause** system call, or during a **wait** system call that does not return immediately due to the existence of a previously stopped or zombie process, the signal-catching function will be executed and then the interrupted system call will return a -1 to the calling process with **errno** set to EINTR.

Note: The **SIGKILL** signal cannot be caught.

Warning: The **signal** system call does not check the validity of the *action* parameter. If it points to a location outside of the process's allocated address space, then the process receives a memory fault when the system attempts to call the signal handler. If *action* points to anything other than a subroutine, the results are unpredictable.

Special Signals

Some signals are handled differently from those described previously. These signals are:

SIGFPE	8*+	Arithmetic exception, integer divide by 0, or floating point exception
SIGDANGER	7+	The system is likely to "crash" soon.
SIGCLD	18+	Death of a child process
SIGPWR	19+	Power-fail restart (not reset when caught)

On a **SIGFPE** signal, the values in all of the floating-point registers are saved. On any other signal, only the first eight registers are saved.

See the **sys/robust.h** header file for the conditions that can cause the **SIGDANGER** signal. The most likely cause is a shortage of paging space (**PGSDANGER**). Also see the **pslotwarn, pslotkill,** and **pslotpanic** keywords in "master" on page 4-98.

For **SIGDANGER** and **SIGPWR**, the actions prescribed by the *action* parameter are as follows:

SIG_DFL The signal is ignored.

SIG_IGN The signal is ignored.

function address The signal-catching function pointed to by *action* is called.

For **SIGCLD**, the actions prescribed by the *action* paramer are as follows:

SIG_DFL The signal is ignored.

SIG_IGN The signal is ignored. Also, the child processes of the calling process do not create zombie processes when they terminate. (See "exit, _exit" on page 2-40 for more information about zombie processes.)

function address The signal-catching function pointed to by *action* is called. When the signal-catching function returns, another **SIGCLD** signal is sent to the process if any zombie child processes remain to be waited for. Therefore, the **SIGCLD** signal-catching function must issue a **wait** system call to eliminate the zombies, or an infinite loop will occur.

The setting of the *action* for the **SIGCLD** signal affects the **wait** and **exit** system calls in the following ways:

wait If the *action* value of **SIGCLD** is set to **SIG_IGN** and a **wait** system call is executed, the **wait** blocks until all of the child processes of the calling process terminate. It then returns a value of -1 with **errno** set to **ECHILD.**

exit If, in the parent of the exiting process, the *action* value of **SIGCLD** is set to **SIG_IGN**, then the exiting process does not create a zombie process.

When processing a pipeline, the shell makes the last process in the pipeline the parent of the preceding processes. A process that can be piped into in this manner, and thus become the parent of other processes, should not set **SIGCLD** to be caught. Otherwise, it will receive unexpected **SIGCLD** signals.

After a **fork** system call, the child process inherits all signals from its parent.

The **exec** system calls reset all caught signals to the default action. Signals that cause the default action continue to do so. Ignored signals continue to be ignored.

Return Value

Upon successful completion, *signal* returns the previous value of *action* for the specified signal *sig*. Otherwise, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **signal** system call fails if the following is true:

EINVAL The *sig* parameter is not a valid signal number, or it is **SIGKILL**.

Related Information

In this book: "acct" on page 2-11, "exit, _exit" on page 2-40, "kill" on page 2-60, "pause" on page 2-94, "ptrace" on page 2-102, "sigblock" on page 2-143, "sigpause" on page 2-150, "sigsetmask" on page 2-152, "sigstack" on page 2-154, "sigvec" on page 2-156, "umask" on page 2-169, "wait" on page 2-182, "setjmp, longjmp" on page 3-332, and "core" on page 4-39.

sigpause

Purpose

Atomically releases blocked signals and waits for an interrupt.

Syntax

int sigpause (*sigmask***)**
int *sigmask***;**

Description

The **sigpause** system call sets the process's signal mask to the value of the *sigmask* parameter and then and waits for a signal to arrive. Upon return, the previous signal mask is restored. The **sigpause** system call terminates by being interrupted, returning -1, and setting **errno** to **EINTR**.

The **sigpause** system call sets the signal mask and waits for an interrupt as one ***atomic operation***. This means that signals cannot occur between the operations of setting the mask and waiting for a signal. If a program invokes the **sigsetmask** and **pause** system calls separately, then a signal that occurs between these system calls might not be noticed by **pause**.

In normal usage, a signal is blocked by using the **sigblock** system call at the beginning of a critical section. The process then determines whether there is work for it to do. If no work is to be done, then the process waits for work by calling **sigpause** with the mask previously returned by **sigblock**.

Return Value

If the signal is caught by the calling process and control is returned from the signal handler, then the calling process resumes execution after the **sigpause** system call, which always returns a value of -1 and sets **errno** to **EINTR**.

Related Information

In this book: “pause” on page 2-94, “sigblock” on page 2-143, “signal” on page 2-145, “sigsetmask” on page 2-152, and “sigvec” on page 2-156.

sigsetmask

Purpose

Sets the current signal mask.

Syntax

int sigsetmask (*mask***)**
int *mask***;**

Description

The **sigsetmask** system call sets the current signal mask as specified by the *mask* parameter. The signal mask determines which signals is blocked from delivery to the process. Signal *i* is blocked if the *i*-th bit in *mask* is a 1.

Typically, you would use the **sigblock** system call to block signals during a critical section of code and then use the **sigsetmask** system call to restore the mask to the previous value returned by the **sigblock** system call.

The **sigsetmask** system call does not allow **SIGKILL** to be blocked. If a program attempts to block **SIGKILL**, **sigsetmask** gives no indication of the error.

Return Value

Upon successful completion, the previous set of masked signals is returned.

Example

To set the signal mask to block only **SIGINT** from delivery:

```
#include <signal.h>

int prevmask;
  . . .
prevmask = sigsetmask (1 << (SIGINT - 1));
```

Related Information

In this book: "kill" on page 2-60, "signal" on page 2-145, "sigvec" on page 2-156, "sigblock" on page 2-143, and "sigpause" on page 2-150.

sigstack

Purpose

Sets and gets signal stack context.

Syntax

#include <signal.h>

int sigstack (*instack*, *outstack*)
struct sigstack **instack*, **outstack*;

Description

The **sigstack** system call defines an alternate stack on which signals are to be processed.

If the value of the *instack* parameter is nonzero, then it points to a **sigstack** structure, which has the following members:

```
caddr_t  ss_sp;
int      ss_onstack;
```

The value of *instack*->**ss_sp** specifies the stack pointer of the new signal stack. Since stacks grow from numerically greater addresses to lower ones, the stack pointer passed to the **sigstack** system call should point to the numerically high end of the stack area to be used. *instack*->**ss_onstack** should be set to 1 if the process is currently executing on that stack; otherwise, it should be 0.

If the value of the *outstack* parameter is nonzero, then it points to a **sigstack** structure into which the **sigstack** system call stores the current signal stack state.

If the value of the *instack* parameter is 0 (that is, a **NULL** pointer), then the signal stack state is not set. If the value of the *outstack* parameter is 0, then the previous signal stack state is not reported.

When a signal occurs whose handler is to run on the signal stack, the system checks to see if the process is already executing on that stack. If so, then it continues to do so even after the handler returns. If not, then the signal handler runs on the signal stack, and the original stack is restored when the handler returns.

Use the **sigvec** system call to specify whether or not a given signal's handler routine is to run on the signal stack.

Warning: A signal stack does not automatically increase in size as a normal stack does. If the stack overflows, unpredictable results may occur.

Return Value

Upon successful completion, a value of 0 is returned. If the **sigstack** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **sigstack** system call fails and the signal stack context remains unchanged if the following is true:

EFAULT The *instack* or *outstack* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "signal" on page 2-145, "sigvec" on page 2-156, and "setjmp, longjmp" on page 3-332.

sigvec

Purpose

Selects enhanced signal facilities.

Syntax

#include <sys/signal.h>

int sigvec (*sig*, *invec*, *outvec*)
int *sig*;
struct sigvec *invec, *outvec;

Description

The **sigvec** system call allows the user to select standard or enhanced signal-handling facilities. Like the **signal** system call, it sets the action to take upon the receipt of a signal, but it also sets additional features.

The **sigvec** system call assigns a handler for a specific signal. If the *invec* parameter is nonzero, it points to a **sigvec** structure that specifies a handler routine and mask to be used when delivering the specified signal. The **sigvec** structure has the following members:

```
int  (*sv_handler) ( );
int  sv_mask;
int  sv_onstack;
```

If the **SIG_STK** bit of **sv_onstack** is set, then the system runs the handler on the signal stack specified by the **sigstack** system call. If this bit is not set, then the handler executes on the stack of the interrupted process. If the **SIG_STD** bit of **sv_onstack** is set, then standard signal processing is used. If this bit is not set, then enhanced signal processing is used.

The default action for a signal can be reinstated by setting **sv_handler** to **SIG_DFL**. If **sv_handler** is set to **SIG_IGN**, then the signal is ignored, and pending instances of the signal are discarded. See “signal” on page 2-145 for a detailed description of the default signal actions.

If the *outvec* parameter is nonzero, then the previous handling information for the signal is stored in the **sigvec** structure pointed to by *outvec*.

If the value of the *invec* parameter is 0 (that is, a **NULL** pointer), then the signal handler information is not set. If the value of the *outvec* parameter is 0, then the previous signal handler information is not reported.

Once a signal handler is assigned, it remains assigned until another **sigvec**, **signal**, or **exec** system call is made.

Warning: The **sigvec** system call does not check the validity of the **sv_handler** pointer. If it points to a location outside of the process's allocated address space, then the process receives a memory fault when the system attempts to call the signal handler. If **sv_handler** points to anything other than a subroutine, then the results are unpredictable.

The signal-handler subroutine can be declared as follows:

> *handler* **(***sig***,** *code***,** *scp***)**
> **int** *sig***,** *code***;**
> **struct sigcontext ****scp***;**

The *sig* parameter is the signal number. The *code* parameter is provided only for compatibility with other UNIX-compatible systems, and its value is always 0. The *scp* parameter points to the **sigcontext** structure that is later used to restore the process's previous execution context. The **sigcontext** structure is defined in **signal.h**.

Note: The **sigcontext** structure contains fields for saving the values of the floating-point registers. On a **SIGFPE** signal, the values in all of the registers are saved. On all other signals, only the first eight registers are saved.

Return Value

Upon successful completion, a value of 0 is returned. If the **sigvec** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **sigvec** system call fails and no new signal handler is installed if one of the following occurs:

EFAULT The *invec* or *outvec* parameter points to a location outside of the process's allocated address space

EINVAL The *sig* parameter is not a valid signal number.

EINVAL An attempt was made to ignore or supply a handler for **SIGKILL**.

Related Information

In this book: "kill" on page 2-60, "ptrace" on page 2-102, "sigblock" on page 2-143, "sigpause" on page 2-150, "sigstack" on page 2-154, "sigsetmask" on page 2-152, "sigvec" on page 2-156, and "setjmp, longjmp" on page 3-332.

The **kill** command in *AIX Operating System Commands Reference.*

stat, fstat

Purpose

Gets the status of a file.

Syntax

#include <sys/stat.h>

int stat (*path*, *buf***)**
**char ****path*;
**struct stat ****buf*;

int fstat (*fildes*, *buf***)**
int *fildes*;
**struct stat ****buf*;

Description

The **stat** system call obtains information about the file pointed to by the *path* parameter. Read, write, or execute permission of the named file is not required, but all directories in the path leading to the file must be searchable. The **stat** system call places the information obtained into a structure pointed to by the *buf* parameter.

Use the **fstat** system call to obtain information about an open file pointed to by the *fildes* parameter. The *fildes* parameter is a file descriptor obtained from a successful **open**, **creat**, **dup**, **fcntl**, or **pipe** system call. The **fstat** system call places the information obtained into a structure pointed to by the *buf* parameter.

The **stat** structure pointed to by the *buf* parameter is defined in the **sys/stat.h** header file, and it contains the following members:

```
dev_t   st_dev;    /* ID of the device that contains */
                   /*   a directory entry for this file */
ino_t   st_ino;    /* Inode number */
ushort  st_mode;   /* File mode; see mknod and chmod */
short   st_nlink;  /* Number of links */
ushort  st_uid;    /* User ID of the file's owner */
ushort  st_gid;    /* Group ID of the file's group */
dev_t   st_rdev;   /* ID of device */
                   /*   st_rdev is defined only for */
                   /*   character or block special files */
off_t   st_size;   /* File size in bytes */
time_t  st_atime;  /* Time of last access */
time_t  st_mtime;  /* Time of last data modification */
time_t  st_ctime;  /* Time of last file status change */
                   /* Times are measured in seconds since */
                   /* 00:00:00 GMT, Jan. 1, 1970 */
```

st_dev The device that contains a directory entry for this file. On a nondistributed file system, this is a 32-bit quantity that uses only the low 16-bits to contain the concatenated 8-bit major device number and the 8-bit minor device number. On a distributed system, this is a 32-bit quantity, created by combining a 16-bit connection ID, the 8-bit major device number, and the 8-bit minor device number.

st_uid The user ID of the file's owner. If the file is a remote file, this value can also be one of the two special values **netnoone** or **netsomeone**, as defined in the **/etc/master** file. For remote files, this field contains the user ID after reverse translation. (See *Managing the AIX Operating System* for a discussion of reverse translation.)

st_gid The group ID of the file's owner. If the file is a remote file, this value can also be one of the two special values **netnoone** or **netsomeone**, as defined in the **/etc/master** file. For remote files, this field contains the group ID after reverse translation. (See *Managing the AIX Operating System* for a discussion of reverse translation.)

st_atime The time when file data was last accessed. For remote files, this field contains the time at the server. Changed by the following system calls: **creat**, **mknod**, **pipe**, **utime**, and **read**.

st_mtime The time when data was last modified. For remote files, this field contains the time at the server. Changed by the following system calls: **creat**, **fclear**, **ftruncate**, **mknod**, **open**, **pipe**, **utime**, and **write**.

st_ctime The time when file status was last changed. For remote files, this field contains the time at the server. Changed by the following system calls: **chmod**, **chown**, **creat**, **link**, **mknod**, **pipe**, **unlink**, **utime**, and **write**.

Return Value

Upon successful completion, both the **stat** and the **fstat** system calls return a value of 0. If either the **stat** or the **fstat** system calls fail, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **stat** system call fails if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

EACCES Search permission is denied for a component of the path prefix.

EFAULT The *buf* or *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **stat** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

The **fstat** system call fails if one or more of the following are true:

EBADF *fildes* is not a valid open file descriptor.

EFAULT *buf* points to a location outside of the process's allocated address space.

If Distributed Services is installed on your system, **fstat** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "chmod" on page 2-18, "chown, chownx" on page 2-21, "creat" on page 2-27, "fullstat, ffullstat" on page 2-50.2, "link" on page 2-62, "mknod" on page 2-69, "pipe" on page 2-95, "read, readx" on page 2-106, "time" on page 2-164, "unlink" on page 2-174, "ustat" on page 2-178, "utime" on page 2-180, "write, writex" on page 2-184, "master" on page 4-98, and "stat.h" on page 5-69.

stime

Purpose

Sets the time.

Syntax

int stime (*tp*)
long *tp;

Description

The **stime** system call sets the system's time and date. The *tp* parameter points to the time as measured in seconds from 00:00:00 GMT January 1, 1970.

Return Value

Upon successful completion, a value of 0 is returned. If the **stime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **stime** system call fails if the following is true:

EPERM The effective user ID of the calling process is not superuser.

Related Information

In this book: "time" on page 2-164.

sync

Purpose

Updates the superblock, i-nodes, and delayed blocks.

Syntax

void sync ( )

Description

The **sync** system call causes all information in memory that should be on disk to be written out. The writing, although scheduled, is not necessarily complete upon return from the **sync** system call. Types of information to be written include modified superblocks, modified i-nodes, delayed block I/O, and read-write mapped files.

If Distributed Services is installed on your system, information in memory relating to remote files is scheduled to be sent to the remote node.

The **sync** system call should be used by programs that examine a file system, such as the **df** and **fsck** commands described in *AIX Operating System Commands Reference.*

Related Information

In this book: "fsync" on page 2-48.

The **sync** command in *AIX Operating System Commands Reference.*

time

Purpose

Gets the time.

Syntax

```
long time ((long *) 0)                    long time (tloc)
                                          long *tloc;
```

Description

The **time** system call returns the current time in seconds since 00:00:00 GMT, January 1, 1970.

If the *tloc* parameter is nonzero, the time is also stored in the location to which the *tloc* parameter points.

Warning: The actions of the **time** system call are undefined if the *tloc* parameter points to a location outside of the process′s allocated address space.

Return Value

Upon successful completion, the current time is returned.

Related Information

In this book: "stime" on page 2-162.

times

Purpose

Gets process and child process times.

Syntax

```
#include <sys/types.h>
#include <sys/times.h>

time_t times (buffer)
struct tms *buffer;
```

Description

The **times** system call fills the structure pointed to by the *buffer* parameter with time-accounting information. All time values reported by the **times** system call are in 10ths of a second, unless execution profiling is enabled. When profiling is enabled, **times** reports values in 60ths of a second. (For more information about profiling, see "profil" on page 2-99, "monitor" on page 3-248, and the **cc** and **prof** commands in *AIX Operating System Commands Reference.*)

The **tms** structure is defined in **sys/times.h** and it contains the following members:

```
time_t  tms_utime;
time_t  tms_stime;
time_t  tms_cutime;
time_t  tms_cstime;
```

This information comes from the calling process and each of its terminated child processes for which it has executed a **wait** system call.

tms_utime The CPU time used while executing instructions in the user space of the calling process.

tms_stime The CPU time used by the system on behalf of the calling process.

tms_cutime The sum of the **tms_utimes** and the **tms_cutimes** of the child processes.

tms_cstime The sum of the **tms_stimes** and the **tms_cstimes** of the child processes.

Note: The system measures time by counting clock interrupts. The precision of the values reported by the **times** system call depends on the rate at which the clock interrupts occur.

Return Value

Upon successful completion, the **times** system call returns the elapsed real time, in 10ths of a second (60ths when profiling), since an arbitrary reference time in the past (for example, system start-up time). This reference time does not change from one call of **times** to another. If the **times** system call fails, a -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **times** system call fails if the following is true:

EFAULT The *buffer* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fork" on page 2-46, "profil" on page 2-99, "time" on page 2-164, "wait" on page 2-182, and "monitor" on page 3-248.

The **cc** and **prof** commands in *AIX Operating System Commands Reference.*

ulimit

Purpose

Sets and gets user limits.

Syntax

#include <sys/types.h>

off_t ulimit (*cmd*, *newlimit***)**
int *cmd*;
off_t *newlimit*;

Description

The **ulimit** system call controls process limits. The *cmd* parameter values are:

1 Returns the process's file size limit. The limit is in units of 512-byte blocks and is inherited by child processes. Files of any size can be read.

2 Sets the process's file size limit to the value of the *newlimit* parameter. Any process can decrease this limit, but only a process with an effective user ID of superuser can increase the limit.

3 Returns the maximum possible break value (see "brk, sbrk" on page 2-14).

1004 Sets the maximum possible break value (see "brk, sbrk" on page 2-14). Returns the new maximum break value, which is *newlimit* rounded *up* to the nearest page boundary.

1005 Returns the lowest valid stack address. (Note that stacks grow from high addresses to low addresses.)

1006 Sets the lowest valid stack address. Returns the new minimum valid stack address, which is *newlimit* rounded *down* to the nearest page boundary.

With remote files, the ulimit values of the client, or local, node are used.

Return Value

Upon successful completion, a nonnegative value is returned. If the **ulimit** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ulimit** system call fails and the limit remains unchanged if:

EPERM A process with an effective user ID other than superuser attempts to increase the file size limit.

EINVAL The *cmd* parameter is a value other than 1, 2, or 3.

Example

To increase the size of the stack segment by 2048 bytes, and set `rc` to the new lowest valid stack address:

```
rc = ulimit(1006, ulimit(1005, 0) - 2048);
```

Related Information

In this book: "brk, sbrk" on page 2-14 and "write, writex" on page 2-184.

umask

Purpose

Sets and gets the value of the file creation mask.

Syntax

int umask (*cmask*)
int *cmask*;

Description

The **umask** system call sets the process's file mode creation mask to the value of the *cmask* parameter. Only the low-order 9 bits of the *cmask* parameter and the file mode creation mask are used.

Return Value

Upon successful completion, the previous value of the file mode creation mask is returned.

Related Information

In this book: "chmod" on page 2-18, "creat" on page 2-27, "mknod" on page 2-69, "open" on page 2-90, and "stat.h" on page 5-69.

The **sh** and **umask** commands in *AIX Operating System Commands Reference.*

umount

Purpose

Unmounts a file system.

Syntax

int umount (*dev*)
**char ****dev*;

Description

The **umount** system call unmounts a previously mounted file system contained on the block device (also called a ***special file***) identified by the *dev* parameter. The *dev* parameter is a pointer to a path name.

After the file system is unmounted, the directory upon which the file system was mounted reverts to its ordinary interpretation as a directory.

The **umount** system call can be invoked only by a process whose effective user ID is superuser.

Return Value

Upon successful completion, a value of 0 is returned. If the **umount** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **umount** system call fails if one or more of the following are true:

EPERM	The process's effective user ID is not superuser.
ENOENT	*dev* does not exist.
ENOTBLK	*dev* is not the name of a block special file.
EINVAL	*dev* is not mounted.
EINVAL	*dev* is not local.
EBUSY	A file on the device specified by the *dev* parameter is currently in use.

EFAULT *dev* points to a location outside of the process's allocated address space.

ENXIO *dev* is not currently configured.

Related Information

In this book: "mount" on page 2-71, "uvmount" on page 2-180.3, and "vmount" on page 2-180.5.

The **mount** and **umount** commands in *AIX Operating System Commands Reference.*

uname, unamex

Purpose

Gets the name of the current AIX system.

Syntax

#include <sys/utsname.h>

int uname (*name*)
struct utsname *_name_;

int unamex (*name*)
struct xutsname *_name_;

Description

The **uname** system call stores information identifying the current system in the structure pointed to by the *name* parameter.

The **uname** system call uses the **utsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
char  sysname[SYS_NMLN];
char  nodename[SYS_NMLN];
char  release[SYS_NMLN];
char  version[SYS_NMLN];
char  machine[SYS_NMLN];
```

The **uname** system call returns a null-terminated character string naming the current system in the character array **sysname**. The **nodename** array contains the name that the system is known by on a communications network. The **release** and **version** arrays further identify the system.

The **machine** array identifies the CPU hardware being used. This array contains an eight-character string followed by a terminating null character. The first two characters identify the hardware model. The hardware model identification may be one of the following:

10	IBM 6151
20	IBM 6150

The remaining six characters of the **machine** string specify the unique serial number of the machine. Each digit of the serial number is in the range `'0'` to `'9'` or `'A'` to `'F'`.

The **unamex** system call uses the **xutsname** structure, which is defined in the **sys/utsname.h** file, and it contains the following members:

```
unsigned long  nid;
long           reserved[3];
```

The **xutsname.nid** field is the binary form of the **utsname.machine** field. For local area networks in which a binary node name is appropriate, **xutsname.nid** contains such a name.

Return Value

Upon successful completion, a nonnegative value is returned. If the **uname** or **unamex** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **uname** and **unamex** system calls fail if:

EFAULT The *name* parameter points to a location outside of the process's allocated address space.

Related Information

The **uname** command in *AIX Operating System Commands Reference.*

unlink

Purpose

Removes a directory entry.

Syntax

int unlink (*path***)**
char ****path*;**

Description

The **unlink** system call removes the directory entry specified by the *path* parameter. If Distributed Services is installed on your system, this path can cross into another node.

When all links to a file are removed and no process has the file open, the space occupied by the file is freed and the file ceases to exist. If one or more processes have the file open when the last link is removed, the removal is postponed until all references to the file are closed.

Return Value

Upon successful completion, a value of 0 is returned. If the **unlink** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **unlink** system call fails and the named file is not unlinked if one or more of the following are true:

ENOTDIR A component of the path prefix is not a directory.

ENOENT The named file does not exist.

EACCES Search permission is denied for a component of the path prefix.

EACCES Write permission is denied on the directory containing the link to be removed.

EPERM The named file is a directory and the effective user ID of the process is not superuser.

EBUSY	The entry to be unlinked is the mount point for a mounted file system.
ETXTBSY	The entry to be unlinked is the last link to a pure procedure (shared text) file that is being executed.
EROFS	The entry to be unlinked is part of a read-only file system.
EFAULT	The *path* parameter points to a location outside of the process's allocated address space.
ESTALE	The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **unlink** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EDIST	The server has a release level of Distributed Services that cannot communicate with this node.
EAGAIN	The server is too busy to accept the request.
ESTALE	The file descriptor for a remote file has become obsolete.
EPERM	The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.
ENODEV	The named file is a remote file located on a device that has been unmounted at the server.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
ENOCONNECT	An attempt to establish a new network connection with a remote node failed.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "close" on page 2-25, "link" on page 2-62, and "open" on page 2-90.

The **rm** command in *AIX Operating System Commands Reference.*

usrinfo

Purpose

Gets and sets user information about the owner of the calling process.

Syntax

#include <uinfo.h>

int usrinfo (*cmd*, *buf*, *count***)**
int *cmd*;
char **buf*;
int *count*;

Description

The **usrinfo** system call gets and sets information about the owner of the current process. The information is a sequence of null-terminated *name*=*value* strings. The last string in the sequence is terminated by two successive null characters. A child process inherits the user information of its parent.

The *buf* parameter is a pointer to a user buffer. This buffer is usually **UINFOSIZ** bytes long.

The *count* parameter is the number of bytes of user information to be copied from or to the user buffer.

If the *cmd* parameter is one of the following constants:

GETUINFO Copies up to *count* bytes of user information into the buffer pointed to by the *buf* parameter.

SETUINFO Sets the user information for the process to the first *count* bytes in the buffer pointed to by the *buf* parameter. The effective user ID of the calling process must be superuser to set the user information.

The user information should at minimum consist of three strings that are typically set by the **login** program. These three strings are:

NAME=*username*
UID=*userid*
TTY=*ttyname*

If the process has no terminal, *ttyname* should be null.

Return Value

Upon successful completion, a nonnegative integer giving the number of bytes transferred is returned. If the **usrinfo** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **usrinfo** system call fails if one or more of the following are true:

EPERM The *cmd* parameter is set to **SETUINFO** and the effective user ID of the process is not superuser.

EINVAL The *cmd* parameter is not set to **SETUINFO** or **GETUINFO**.

EINVAL The *cmd* parameter is set to **SETUINFO** and the count parameter is larger than **UINFOSIZ**.

EINVAL The *cmd* parameter is **SETUINFO** and *buf* does not contain a **NAME=** entry.

EFAULT The *buf* parameter points to a location outside of the process's allocated address space.

Related Information

In this book: "getuinfo" on page 3-223.

The **login** command in *AIX Operating System Commands Reference.*

ustat

Purpose

Gets file system statistics.

Syntax

#include <sys/types.h>
#include <ustat.h>

int ustat (*dev*, *buf***)**
dev_t *dev***;**
struct ustat ***buf*;**

Description

The **ustat** system call gets information about the mounted file system identified by device number *dev*. The *dev* parameter is the ID of the device, and it corresponds to the **st_dev** member of the structure returned by the **stat** and **fullstat** system calls. If the high-order bits of the device number are zero, then the device is a local device. If the high-order bits of the device number are nonzero, then the calling node queries the remote node over the connection identified by these 16 bits. See "fullstat, ffullstat" on page 2-50.2, "stat, fstat" on page 2-159, "fullstat.h" on page 5-56.2, and "stat.h" on page 5-69 for more information about the device ID. The information is stored into a structure pointed to by the *buf* parameter.

The **ustat** structure pointed to by the *buf* parameter is defined in the **ustat.h** file, and it contains the following members:

```
daddr_t  f_tfree;          /* Total free blocks */
ino_t    f_tinode;         /* Number of free i-nodes */
char     f_fname[6];       /* File system name */
char     f_fpack[6];       /* File system pack name */
```

Return Value

Upon successful completion, a value of 0 is returned. If the **ustat** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **ustat** system call fails if one or more of the following are true:

EINVAL	*dev* is not the device number of a device containing a mounted file system.
EFAULT	The *buf* parameter points to a location outside of the process's allocated address space.

If Distributed Services is installed on your system, **ustat** can also fail if one or more of the following are true:

EDIST	The server has blocked new inbound requests.
EDIST	Outbound requests are currently blocked.
EAGAIN	The server is too busy to accept the request.
ESTALE	*dev* identifies a remote device that can no longer be reached.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "fullstat, ffullstat" on page 2-50.2, "stat, fstat" on page 2-159, and "fs" on page 4-74.

utime

Purpose

Sets file access and modification times.

Syntax

#include <unistd.h>

int utime (*path*, *times*)
char *path;
struct utimbuf *times;

Description

The **utime** system call sets the access and modification times of the file pointed to by the *path* parameter to the value of the *times* parameter. If Distributed Services is installed on your system, this path can cross into another node.

If the *times* parameter is **NULL**, the access and modification times of the file are set to the current time. If the file is a remote file, the current time at the remote node, rather than the local node, is used. The effective user ID of the process must be the same as the owner of the file or must have write permission in order to use the **utime** system call in this manner.

If the *times* parameter is not **NULL**, it is a pointer to a **utimbuf** structure and the access and modification times are set to the values contained in the designated structure, regardless of whether or not those times correlate with the current time. For remote files, if the **utime** system call is used in this way, the file's times may be different from the time at the remote node. Only the owner of the file or superuser can use the **utime** system call this way.

The **utimbuf** structure pointed to by the *times* parameter is defined in the **unistd.h** file, and it contains the following members.

```
time_t actime;    /* Date and time of last access */
time_t modtime;   /* Date and time of last modification */
```

The times in this structure are measured in seconds since 00:00:00 GMT, January 1, 1970.

Return Value

Upon successful completion, a value of 0 is returned. If the **utime** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **utime** system call fails if one or more of the following are true:

ENOENT The named file does not exist.

ENOTDIR A component of the path prefix is not a directory.

EACCES Search permission is denied by a component of the path prefix.

EPERM The effective user ID is not superuser or the owner of the file and the *times* parameter is not **NULL**.

EACCES The effective user ID is not superuser or the owner of the file, the *times* parameter is **NULL**, and write access is denied.

EROFS The file system containing the file is mounted read-only.

EFAULT The *times* or *path* parameter points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **utime** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "stat, fstat" on page 2-159.

uvmount

Purpose

Unmounts a device, directory, or file.

Syntax

int uvmount (*stubpath***)**
char ****stubpath*;**

Description

The **uvmount** system call unmounts the device, directory, or file that is mounted on *stubpath*. The *stubpath* parameter points to a path name. If Distributed Services is installed on your system, this path can lead to a remote node.

To issue a **uvmount** system call, this process's effective user ID must be superuser, or the process must own the *stubpath* directory and have write permission on the parent directory of the *stubpath*.

After the **uvmount** system call has been completed, *stubpath* reverts to its previous interpretation as a directory or file.

Return Value

Upon successful completion, the **uvmount** system call returns a value of 0. If the **uvmount** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **uvmount** system call fails if one or more of the following are true:

EBUSY	A device that is still in use is being unmounted.
EPERM	The process's effective user ID is not superuser nor is this process the owner of the *stubpath* directory, with write permission on the parent directory.
EINVAL	There is nothing mounted on *stubpath*.
EFAULT	*stubpath* points to a location outside of the process's allocated address space.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **uvmount** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV The named file is a remote file located on a device that has been unmounted at the server.

ENOMEM Either this node or the server does not have enough memory available to service the request.

ENOCONNECT An attempt to establish a new network connection with a remote node failed.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "mntctl" on page 2-70.2, "mount" on page 2-71, "umount" on page 2-170, "vmount" on page 2-180.5, and "master" on page 4-98.

The **mount** and **umount** commands in *AIX Operating System Commands Reference.*

vmount

Purpose

Mounts a directory or a regular file.

Syntax

#include <sys/vmount.h>

int vmount (*path*, *stubpath*, *mflag*, *type*, *info*, *size***)**
char ****path*, ******stubpath*;**
int *mflag***;**
unsigned int *type***;**
unsigned long ****info*;**
int *size***;**

Description

The **vmount** system call provides, in addition to the function of the **mount** system call, the following types of mounts:

- Local file or directory over local file or directory
- Local file or directory over remote file or directory
- Remote file or directory over local file or directory
- Remote file or directory over remote file or directory.

A directory can only be mounted over a directory, and a file can only be mounted over a file.

A process must have an effective user ID of superuser to use the **vmount** system call.

The **vmount** system call mounts the file or directory identified by the *path* parameter on the file or directory identified by the *stubpath* parameter. The *path* and *stubpath* parameters are both pointers to null-terminated path names. For the **mntctl** system call to be useful in reporting current virtual mounts, these paths should both be absolute path names, beginning at the root directory with a / (slash).

The *type* parameter identifies the type of file system that is being mounted with the *path* parameter. The setting of this parameter also determines whether the *info* and *size* parameters are used or ignored. The following values are permitted for *type*:

MNT_AIX Indicates that *path* identifies a local entity, and the system can determine the type of the entity by examining it. For this *type*, the *info* and *size* parameters are ignored by the system call.

When mounting any local device, file, or directory, specify this value.

MNT_DS Indicates that *path* identifies an entity in a remote node. For this *type*, *info* points to the node ID of the remote node, and *size* is **sizeof(long)**.

When mounting any remote file or directory, specify this value.

The *mflag* parameter defines various characteristics of the object to be mounted. Possible values are:

MNT_READONLY Indicates that the object is read-only, and write access is not allowed. If this value is not specified, writing is permitted according to individual file accessibility.

MNT_REMOVABLE Indicates that the object is a removable file system. Whenever there are no active references to files or directories on the file system, the operating system forgets the content and structure of the file system. The user can remove the medium and replace it with a different file system. All future references to *stubpath* will refer to the file system on the new medium.

Return Value

Upon successful completion, the **vmount** system call returns a value of 0. If the **vmount** system call fails, a value of -1 is returned, and **errno** is set to indicate the error.

Diagnostics

The **vmount** system call fails and the file or directory is not mounted if one or more of the following are true:

EBUSY *stubpath* is currently busy. Possible causes include:

- The kernel's mount table is full.
- The *path* parameter indicates a device that is currently mounted.

EBUSY The special file to be mounted, defined by the *path* parameter, is already open for writing.

ENOTBLK The object to be mounted is not a file, directory, or device.

EFAULT The *info*, *stubpath*, or *path* parameter points to a location outside of the process's allocated address space.

E2BIG The number of characters in *path* is greater than **MAXPATH**.

EPERM This process is not the owner of the *stubpath* directory.

EPERM This process does not have write permission on the parent directory of *stubpath.*

EPERM This process does not have an effective user ID of superuser and is attempting to mount a device.

ENOTDIR A device is being mounted over a non-directory or over a remote directory.

ENOTDIR A mount of a remote device has been attempted.

ENOTDIR A component of the path prefix is not a directory.

ENOTDIR *stubfile* and *path* are not both files or both directories.

EINVAL For a remote mount, the *info* parameter did not point to a valid node ID.

EINVAL A remote mount is requested, but *stubpath* is not a full path name beginning with / (slash).

EINVAL *type* is not a recognized file system type.

EINVAL *path* indicates a device that does not contain a recognizable file system.

EINVAL A *type* of **MNT_DS** was specified, but the local node ID was specified in *info.*

EINVAL A remote mount is requested, but *path* is not local to the server.

EINVAL *path* indicates a device that contains a corrupted file system.

ESTALE The process's root or current directory is located in a virtual file system that has been unmounted.

If Distributed Services is installed on your system, **vmount** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EDIST The server has a release level of Distributed Services that cannot communicate with this node.

EAGAIN The server is too busy to accept the request.

ESTALE The file descriptor for a remote file has become obsolete.

EPERM The translate tables of the server did not contain any entry for either the effective user ID or effective group ID of the calling process.

ENODEV	The named file is a remote file located on a device that has been unmounted at the server.
ENOMEM	Either this node or the server does not have enough memory available to service the request.
ENOCONNECT	An attempt to establish a new network connection with a remote node failed.
EBADCONNECT	An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "mntctl" on page 2-70.2, "mount" on page 2-71, "umount" on page 2-170, "uvmount" on page 2-180.3, and "master" on page 4-98.

The **mount** and **umount** commands in *AIX Operating System Commands Reference.*

wait

Purpose

Waits for a child process to stop or terminate.

Syntax

int wait (*stat_loc***)**
int ****stat_loc*;**

int wait ((int *) 0)

Description

The **wait** system call suspends the calling process until it receives a signal that is to be caught, or until any one of the calling process's child processes stops in a trace mode or terminates. **wait** returns without waiting if a child process that hasn't been waited for has already stopped or terminated prior to the call.

If the *stat_loc* parameter is nonzero, 16 bits of information called *status* are stored in the low-order 16 bits of the location pointed to by *stat_loc*. The *status* information can be used to differentiate between stopped and terminated child processes and, if the child process terminated, the *status* information identifies the cause of termination and passes information to the parent process. This is accomplished in the following manner:

- If the child process stopped in a trace mode, then the high-order 8 bits of *status* contain the number of the signal that caused the process to stop and the low-order 8 bits are set equal to 0177 (0x7F).
- If the child process terminated due to an **exit** system call, the low-order 8 bits of *status* are 0 and the high-order 8 bits contain the low-order 8 bits of the parameter that the child passed to the **exit** system call.
- If the child process terminated due to a signal, the high-order 8 bits of *status* are 0 and the low-order 8 bits contain the number of the signal that caused the termination. In addition, if the low-order seventh bit (bit 0200 or 0x80) is set, then a memory image file is produced before **wait** returns.

If a parent process terminates without waiting for its child processes to terminate, the parent process ID of each child process is set to 1. This means the initialization process inherits the child processes.

Note: The effect of the **wait** system call can be modified by the setting of the **SIGCLD** signal. See "Special Signals" on page 2-148 for details.

Warning: The actions of the **wait** system call are undefined if the *stat_loc* parameter points to a location outside of the process's allocated address space.

Return Value

If the **wait** system call returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. If the **wait** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **wait** system call fails and returns without waiting if one or more of the following are true:

ECHILD The calling process has no existing unwaited-for child processes.

EINTR The **wait** system call received a signal.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, "fork" on page 2-46, "pause" on page 2-94, "ptrace" on page 2-102, and "signal" on page 2-145.

write, writex

Purpose

Writes to a file.

Syntax

int write (*fildes*, *buf*, *nbytes*)
int *fildes*;
char **buf*;
unsigned int *nbytes*;

int writex (*fildes*, *buf*, *nbytes*, *ext*)
int *fildes*;
char **buf*;
unsigned int *nbytes*;
int *ext*;

Description

The **write** system call writes the number of bytes specified by the *nbytes* parameter from the buffer specified by the *buf* parameter to the file associated with the *fildes* parameter. If Distributed Services is installed on your system, this file can reside on another node.

The *fildes* parameter is a file descriptor obtained from a **creat**, **open**, **dup**, **fcntl**, or **pipe** system call.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from the **write** system call, the file pointer increments by the number of bytes actually written.

On devices incapable of seeking, writing always takes place starting at the current position. The value of a file pointer associated with such a device is undefined.

When the **O_APPEND** flag of the file status is set, the file pointer is set to the end of the file prior to each write.

If the **write** system call requests that more bytes be written than there is room for, only as many bytes as there is room for are written and the **write** system call returns an integer equal to the number of bytes written. The next attempt to write nonzero number of bytes will fail (except as noted following). The limit reached can be either the ulimit (see "ulimit" on page 2-167) or the end of the physical medium. A partial write is not permitted for the following:

- If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is set, then a **write** to a full pipe (or FIFO) returns a count of 0.

- If the file being written is a pipe (or FIFO) and the O_NDELAY flag of the file flag word is not set, then a **write** to a full pipe (or FIFO) blocks until space becomes available.

If the file to be written supports enforcement mode record locks and all or part of the region to be written is currently locked by another process, then the action taken depends on the setting of the **O_NDELAY** flag:

- If **O_NDELAY** is set, then **write** returns -1 and sets **errno** to **EAGAIN**.
- If **O_NDELAY** is not set, then the calling process blocks until the lock is released.

For more information about record locks, see "lockf" on page 2-64.

If the file has been mapped, the **write** system call writes to a mapped file segment. If the *fildes* file descriptor was used to map the file copy-on-write, then the copy-on-write segment is used. Otherwise, the **write** system call writes to the read-write mapped segment for the file.

Warning: If a process issues a **write** system call to a file that it has not mapped, but that other processes have mapped copy-on-write, then the results are unpredictable. However, if the process first attaches the mapped file in copy-on-write mode with the **shmat** system call, then the **write** to the file is properly reflected in the copy-on-write shared segment.

The **writex** system call performs the same function as **write**, except that it provides communication with character device drivers that require more information or return more status than **write** can handle.

For files, directories, or special files with drivers that do not handle extended operations, the **writex** system call does exactly what the **write** system call does, and the *ext* parameter is ignored.

Each driver interprets the *ext* parameter in a device-dependent way, either as a value or as a pointer to a communication area. The nonextended **write** system call is equivalent to the extended **writex** system call with an *ext* parameter value of 0. Drivers must apply reasonable defaults when the *ext* parameter value is 0.

Return Value

Upon successful completion, the number of bytes actually written is returned. If the **write** or **writex** system call fails, a value of -1 is returned and **errno** is set to indicate the error.

Diagnostics

The **write** and **writex** system calls fail and the file pointer remains unchanged if one or more of the following are true:

EBADF The *fildes* parameter is not a valid file descriptor open for writing.

EAGAIN An enforcement mode record lock is outstanding in the portion of the file that is to be written.

EPIPE An attempt is made to write to a pipe that is not open for reading by any process. A **SIGPIPE** signal is also sent to the calling process.

EFBIG An attempt is made to write a file that exceeds the process's file size limit or the maximum file size (see "ulimit" on page 2-167). If Distributed Services is installed on your system, the file size cannot exceed the client's default file size limit.

EFAULT *buf* points to a location outside of the process's allocated address space.

EDEADLK A deadlock would occur if the calling process were to sleep until the region to be written was unlocked.

EINTR A signal was caught during the **write** system call.

If Distributed Services is installed on your system, **write** can also fail if one or more of the following are true:

EDIST The server has blocked new inbound requests.

EDIST Outbound requests are currently blocked.

EAGAIN The server is too busy to accept the request.

ENOMEM Either this node or the server does not have enough memory available to service the request.

EBADCONNECT An attempt to use an existing network connection with a remote node failed.

Related Information

In this book: "creat" on page 2-27, "dup" on page 2-32, "lockf" on page 2-64, "lseek" on page 2-67, "open" on page 2-90, "pipe" on page 2-95, and "ulimit" on page 2-167, and Appendix C, "Writing Device Drivers."

Chapter 3. Subroutines

About This Chapter

This chapter gives detailed information about the subroutines (also called ***functions***) that are available in standard AIX subroutine libraries. For an explanation of the differences between system calls and subroutines, see the introduction to Chapter 2 on page 2-2. For an explanation of the "Syntax" section of each entry, see "Syntax" on page v. For an explanation of header files, see "Header Files" on page vii.

Each subroutine entry contains a "Library" section that indicates the library where the subroutine is stored. Subroutines are stored in libraries to conserve storage space and to make the program linkage process more efficient. A ***library*** (sometimes called an ***archive***) is a data file that contains copies of a number of individual files and control information that allows them to be accessed individually. See "ar" on page 4-18 and the **ar** command in *AIX Operating System Commands Reference* for more information about libraries.

The libraries that contain the subroutines described in this book are located in the **/usr/lib** directory. By convention, all of them have names of the form **lib*name*.a**, where *name* identifies the specific library.

You do not need to do anything special to use subroutines from the Standard C Library (**libc.a**) or the Run-time Services Library (**librts.a**). The **cc** command automatically searches these libraries for subroutines that a program needs. However, if you use subroutines from another library, you must tell the compiler to search that library. If your program uses subroutines from the library **lib*name*.a**, compile your program with the flag **-l*name***. The following example compiles the program `myprog.c`, which uses subroutines from the **libdbm.a**:

```
cc myprog.c -ldbm
```

You can specify more than one **-l** flag, but they must be specified *after* any other flags. See the **cc** command in *AIX Operating System Commands Reference* for details.

The libraries discussed in the book are:

- Curses Library (**libcurses.a**)
- Database Library (**libdbm.a**)
- DOS Services Library (**libdos.a**)
- Extended Curses Library (**libcur.a**)
- Graphics Libraries (**libplot.a**, **libprint.a**, **lib300.a**, and others)
- IPC Library (**libipc.a**)
- Math Library (**libm.a**)
- Object File Access Routine Library (**libld.a**)
- Programmers Workbench Library (**libPW.a**)
- Run-time Services Library (**librts.a**)
- Standard C Library (**libc.a**)
- Standard I/O Package (**libc.a**)
- Usability Services Library (**libpanels.a**).

The Standard I/O Package subroutines are actually contained in the Standard C Library (**libc.a**). These subroutines implement a buffered I/O system on top of the basic I/O provided by the system calls. For more information about these subroutines, see "standard i/o library" on page 3-342.

Note: A few of the subroutines are stored in libraries that may not be included in your system configuration. If the likage editor (**ld**, which is called by the **cc** command) gives you an error message indicating that it cannot find one of the subroutines, then check Appendix E, "Component Cross Reference." The subroutines listed there are shipped with programs or licensed programs that must be installed separately.

a64l, l64a

Purpose

Converts between long integers and base-64 ASCII strings.

Library

Standard C Library (**libc.a**)

Syntax

long a64l (*s***)**
char ****s*;**

char *l64a (*l***)**
long *l***;**

Description

The **a64l** and **l64a** subroutines maintain numbers stored in base-64 ASCII characters. This is a notation in which long integers are represented by up to 6 characters, each character representing a digit in a base-64 notation.

The following characters are used to represent digits:

.	represents	0.
/	represents	1.
0—9	represent	2—11.
A—Z	represent	12—37.
a—z	represent	38—63.

The **a64l** subroutine takes a pointer to a null-terminated character string containing a value in base-64 representation and returns the corresponding **long** value. If the string pointed to by the *s* parameter contains more than 6 characters, the **a64l** subroutine uses only the first 6.

Conversely, the **l64a** subroutine takes a **long** parameter and returns a pointer to the corresponding base-64 representation. If the *l* parameter is 0, then the **l64a** subroutine returns a pointer to a null string.

The value returned by **l64a** is a pointer into a static buffer, the contents of which are overwritten by each call.

abort

Purpose

Generates an IOT fault to terminate the current process.

Library

Standard C Library (**libc.a**)

Syntax

int abort ( )

Description

The **abort** subroutine causes a **SIGIOT** signal to be sent to the current process. This usually terminates the process and produces a memory dump.

It is possible for the **abort** subroutine to return control if **SIGIOT** is caught or ignored. In this case, **abort** returns the value returned by the **kill** system call.

If **SIGIOT** is neither caught nor ignored, and if the current directory is writable, then the **abort** subroutine produces a memory dump in a file named **core** in the current directory. The shell then displays the message:

```
abort - core dumped
```

Related Information

In this book: "exit, _exit" on page 2-40, "kill" on page 2-60, and "signal" on page 2-145.

The **adb** command in *AIX Operating System Commands Reference.*

abs

Purpose

Returns the absolute value of an integer.

Library

Standard C Library (**libc.a**)

Syntax

int abs (*i***)**
int *i***;**

Description

The **abs** subroutine returns the absolute value of its integer operand.

Note: A two's-complement integer can hold a negative number whose absolute value is too large for the integer to hold. When given this largest negative value, the **abs** subroutine returns the same value.

Related Information

In this book: "floor, ceil, fmod, fabs" on page 3-167.

assert

Purpose

Verifies a program assertion.

Library

Standard C Library (**libc.a**)

Syntax

#include <assert.h>

void assert (*expression***)**
int *expression*;

Description

The **assert** macro puts diagnostics into a program. If *expression* is false (zero), then **assert** writes the following message on the standard error output and aborts the program:

```
    Assertion failed: expression, file filename, line linenum
```

In the error message, *filename* is the name of the source file and *linenum* is the source line number of the **assert** statement.

If you compile a program with the preprocessor option **-DNDEBUG**, or with the preprocessor control statement **#define NDEBUG** ahead of the **#include <assert.h>** statement, assertions will not be compiled into the program.

Related Information

In this book: "abort" on page 3-5.

The **cpp** command in *AIX Operating System Commands Reference.*

atof, strtod

Purpose

Converts an ASCII string to a floating-point number.

Library

Standard C Library (**libc.a**)

Syntax

double atof (*nptr***)**
**char ****nptr*;

double strtod (*nptr*, *ptr***)**
**char ****nptr*, *******ptr*;

Description

The **atof** and **strtod** subroutines convert a character string, pointed to by the *nptr* parameter, to a double-precision floating-point number. The first unrecognized character ends the conversion.

These subroutines recognize a character string when the characters appear in the following order:

1. An optional string of white-space characters
2. An optional sign
3. A string of digits optionally containing a decimal point
4. An optional **e** or **E** followed by an optionally signed integer.

If the string begins with an unrecognized character, **atof** and **strtod** return the value 0.

If the value of *ptr* is not **(char **) NULL**, then a pointer to the character that terminated the scan is stored in **ptr*. If an integer cannot be formed, **ptr* is set to *nptr*, and 0 is returned.

If the correct return value overflows, **atof** and **strtod** return **INF**. On underflow, **atof** and **strtod** return 0.

The **atof (***nptr***)** subroutine call is equivalent to **strtod (***nptr*, **(char **) NULL)**.

The **atof** and **strtod** subroutines perform conversions to a floating-point number. See "strtol, atol, atoi" on page 3-347 for information on conversions to integers.

Related Information

In this book: "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325 and "strtol, atol, atoi" on page 3-347.

bessel: j0, j1, jn, y0, y1, yn

Purpose

Computes Bessel functions.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>

double j0 (x)
double x;

double j1 (x)
double x;

double jn (n, x)
int n;
double x;

double y0 (x)
double x;

double y1 (x)
double x;

double yn (n, x)
int n; double x;
```

Description

The **j0** and **j1** subroutines return Bessel functions of *x* of the first kind, of orders 0 and 1, respectively. **jn** returns the Bessel function of *x* of the first kind of order *n*.

The **y0** and **y1** subroutines return the Bessel functions of *x* of the second kind, of orders 0 and 1, respectively. **yn** returns the Bessel function of *x* of the second kind of order *n*. The value of *x* must be positive.

Non-positive parameters cause **y0**, **y1**, and **yn** to return the value **HUGE**, to set **errno** to **EDOM**, and to write a message to the standard error output indicating a DOMAIN error.

Parameters that are too large in magnitude cause **j0**, **j1**, **y0**, and **y1** to return as much of the result as possible, to set **errno** to **ERANGE**, and to write a message to the standard error output indicating a PLOSS error.

You can change these error-handling procedures with the **matherr** subroutine.

Related Information

In this book: "matherr" on page 3-238.

bsearch

Purpose

Performs a binary search.

Library

Standard C Library (**libc.a**)

Syntax

#include <search.h>

char *bsearch ((char *)*key*, **(char *)***base*, *nel*, **sizeof (****key*), *compar***)**
unsigned int *nel*;
int (**compar***) ( );**

Description

The **bsearch** subroutine is a binary search routine generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.1, Algorithm B.* It returns a pointer into a table indicating where a datum is found.

The table must already be sorted in increasing order according to the provided comparison function *compar*. The *key* parameter points to the datum to be sought in the table. The *base* parameter points to the element at the base of the table. The *nel* parameter is the number of elements in the table. The *compar* parameter is a pointer to the comparison function, which is called with two parameters that point to the elements being compared.

The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, *compar* must return a value less than 0.
- If the first parameter is equal to the second parameter, *compar* must return 0.
- If the first parameter is greater than the second parameter, *compar* must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

* Reading, Massachusetts: Addison-Wesley, 1981.

The pointers *key* and *base* should be of type pointer-to-element, and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Return Value

If the key is found in the table, the **bsearch** returns a pointer to the element found. If the key cannot be found in the table, then **bsearch** returns the value **NULL**.

Related Information

In this book: "hsearch, hcreate, hdestroy" on page 3-227, "lsearch" on page 3-234, "qsort" on page 3-315, and "tsearch, tdelete, twalk" on page 3-364.

cfgabdds

Purpose

Builds and initializes a Define_Device Structure, and then issues a Define_Device SVC.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <sys/bioca.h>

int cfgabdds (*argc*, *argv*, *ptr*, *len***)**
int *argc***;**
char **argv***[ ];**
char **ptr***;**
int *len***;**

Description

A customize helper program must issue the **Define_Device** SVC for each VRM device driver. The **cfgabdds** subroutine is provided for use in customize helper programs to simplify the task of building the **Define_Device** structure and issuing the SVC. See *AIX Operating System Programming Tools and Interfaces* for more detailed information about using this subroutine.

The **cfgabdds** subroutine takes the following parameters:

argc The number of elements in the *argv* array passed to the customize helper by the calling program, which is normally the **vrmconfig** comand.

argv An array of parameters passed to the customize helper by the calling program, which is normally the **vrmconfig** command.

ptr A pointer to the initialized structure of device-dependent information for the device being added to the system. This structure does not include hardware or block I/O device characteristics. If the device requires no device-dependent information, then set this parameter to **NULL**.

len The length in bytes of the structure pointed to by the *ptr* parameter. If the device requires no device-dependent information, then set this parameter to 0.

The customize helper must pass *argc* and *argv* to **cfgabdds** without modification.

Return Value

Upon successful completion, the **cfgabdds** subroutine returns the value **VRCSUCC**. If **cfgabdds** fails, then one of the following values is returned:

VRCKCORP A configuration file is not in attribute file format.

VRCKUNXF An error was returned from the SVC that configures the AIX device driver.

VRCKIOPT An invalid option was specified from **vrmconfig**.

VRCKNOSP No storage space available. The **malloc** subroutine returned a **NULL** value.

VRCKIARG One or more parameters are invalid.

VRCKSTNF **kaf_file** keyword value not found in the *argv* passed from **vrmconfig**.

VRCKASNF **kaf_file** keyword value not found in the *argv* passed from **vrmconfig**.

VRCKANOP Unable to open **kaf** file.

VRCKYINF Keyword not found in input file.

VRCKYWNF Keyword not found in **kaf** file.

VRCKNONE No processing required by library routine.

VRCKNOBY Length of device-dependent information exceeds the limit of the **Define_Device** Structure. (The total space allowed for entire DDS is 500 bytes.)

Related Information

In this book: Appendix C, "Writing Device Drivers."

AIX Operating System Programming Tools and Interfaces.

cfgadev

Purpose

Adds a device.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg01.h>

int cfgadev (*master*, *system*, *xstanza*, *vstanza*, *dstanza*, *vflag*, *cflag***)**
char **master*, **system*, **xstanza*, **vstanza*, **dstanza*;
int *vflag*, *cflag*;

Description

The **cfgadev** subroutine adds information about devices and device drivers to the system configuration.

The *master* parameter points to the full path name of the **master** file. The *system* parameter points to the full path name of the **system** file. These files are usually **/etc/master** and **/etc/system**, respectively.

The *xstanza*, *vstanza*, and *dstanza* parameters point to buffers that contain the text of attribute file stanzas. Any one or two of these parameters can be **NULL** pointers, indicating that a stanza of that type is not to be added, but at least one of them must point to a stanza buffer.

The *xstanza* parameter points to an AIX device driver stanza to be added to the *master* file. If the major device number is missing from the stanza, then the **cfgadev** subroutine generates a new one, beginning with decimal 20.

The *vstanza* parameter points to a VRM device driver stanza to be added to the *master* file. If the IOCN is missing from the stanza, then the **cfgadev** subroutine generates a new one, beginning with decimal 1024.

The *dstanza* parameter points to a device stanza to be added to the *system* file. If the IODN is missing from the stanza, then the **cfgadev** subroutine generates a new one, beginning with decimal 12000. It also generates a minor device number if only the prefix is supplied or if the value is not unique.

The *vflag* parameter is either 1 (for "yes") or 0 (for "no"), indicating whether to execute the **vrmconfig** command after the device stanza is added. If the *vflag* parameter is 1, then **cfgadev** executes the **vrmconfig** command with the **-a** *stname* flag, where *stname* is the name of the device stanza. The **vrmconfig** command then processes this stanza for driver addition and produces a shell procedure. The **cfgadev** subroutine then runs this shell procedure, which creates the special file **/dev/***stname*, where *stname* is the name of the device stanza in the *system* file. If the **vrmconfig** command returns an error, then all stanzas that were added to the *master* and *system* files are deleted.

The *cflag* parameter is either 1 (for "yes") or 0 (for "no"), indicating whether to attempt to associate the VRM device driver stanza with one that has been defined previously. If the *cflag* parameter is 1, then the **cfgadev** subroutine gets the VRM device driver stanza associated with the device being added. If this VRM device driver stanza contains a **code** keyword, then **cfgadev** searches through the *master* file for other stanzas that contain the same **code** keyword value and that are associated with a device stanza, in the *system* file, with the same **dtype** value. If another stanza is found and if that stanza is defined before the stanza for the device being added, then the **cfgadev** subroutine replaces the **code** keyword with a **copy** keyword to copy the stanza that was found. If the VRM device driver stanza contains a **copy** keyword, then **cfgadev** verifies that the stanza it is copying is defined before the VRM device driver stanza of the device being added.

If the device stanza pointed to by the *dstanza* parameter contains the **admgr** keyword, then its value specifies the name of the device manager's stanza in the *system* file. The new device is added to the **vdmgr** keyword value list in the device manager's stanza.

If the device stanza pointed to by the *dstanza* parameter contains the **specproc** keyword, then the program specified by the value of this keyword is executed to perform any special processing required when adding this device. The value of the **specproc** keyword must be the full path name of an executable file. The following arguments are passed to the program using the **argv** mechanism described in "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34. All of them are passed as character strings.

argv[0]	The full path name of the special-processing program
argv[1]	The full path name of the *master* file
argv[2]	The full path name of the *system* file
argv[3]	The name of the device stanza
argv[4]	The character string "a", indicating addition.

If the special processing program fails, then the device is still added to the system, but additional steps may be required before it can be used.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgadev** subroutine fails, then one of the following values is returned:

CFG_BEMP The *xstanza*, *vstanza*, and *dstanza* parameters are all **NULL** pointers.

CFG_BFIC An input stanza is incomplete, or necessary information is missing.

CFG_BFNA A failure occured while adding a stanza to the *master* or *system* file.

CFG_BFSM An input stanza buffer can not be updated because the buffer is too small.

CFG_CFLI The *cflag* parameter is 1, but the device stanza pointed to by the *dstanza* parameter contains **nocopy** = **true**. Or, the VRM device driver stanza for the new device contains a **copy** keyword, but the stanza that it copies is not defined before it in the **/etc/master** file.

CFG_CLSE An error was detected while trying to close a file.

CFG_FCOR The *master* or *system* file is set up incorrectly.

CFG_MALF Memory allocation failed because of insufficient space.

CFG_MAXM The maximum number of minor device numbers has been reached for the driver associated with the device being added.

CFG_MGRF A failure occurred while updating the device manager's stanza.

CFG_MPRE The prefix of the device's minor number is neither **b** nor **c**.

CFG_OPNE An error was detected while trying to open a file.

CFG_SLPF Special processing failed. The device was added but may require some additional steps before it can be used.

CFGT_VLNG An IOCN value, an IODN value, or a major device number could not be generated to complete an input stanza.

CFG_VCFG The **vrmconfig** command failed.

File

/etc/specials

Related Information

In this book: "attributes" on page 4-20, "master" on page 4-98, and "system" on page 4-139.

The **vrmconfig** command in *AIX Operating System Commands Reference.*

cfgamni

Purpose

Adds a minidisk.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg02.h>

int cfgamni (*sysstanza*, *fsstanza*, *use*, *loc***)**
**char ****sysstanza*, **fsstanza*;
int *use*, *loc*;

Description

The **cfgamni** subroutine creates minidisks. The necessary steps to accomplish this include adding stanzas to the **/etc/system** and the **/etc/filesystems** files, updating coprocessor stanzas, executing the **vrmconfig** command to define the minidisk, and issuing AIX subroutine calls to make the file system usable. The calling process must have an effective ID of superuser.

The *sysstanza* and *fsstanza* parameters point to buffers that contain the text of attribute file stanzas. The *sysstanza* parameter points to a minidisk stanza that is to be added to the **/etc/system** file. The *fsstanza* parameter points to a minidisk stanza that is to be added to the **/etc/filesystems** file.

The *use* parameter specifies how the minidisk is used. The values allowed for this parameter are:

PARTUNIX	AIX file system partition
PARTCOPR	Coprocessor partition
PARTOTHR	A partition for some other use.

The *loc* parameter specifies the approximate location on the physical disk where the minidisk should reside. The values allowed for this parameter are:

PARTLOCH	At the high end of the fixed disk
PARTLOCM	Near the center of the fixed disk

PARTLOCL Near the beginning of the fixed disk.

After the **cfgamni** subroutine completes successfully, the **/etc/system** file contains a stanza for the new minidisk, and the minidisk has been added to the system.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgamni** subroutine fails, then one of the following values is returned:

CFG_NSID The calling process's effective user ID is not superuser.

CFG_USZF The format of a stanza is incorrect.

CFG_MAXP The maximum number of minidisks are already defined.

CFG_VRMF The **vrmconfig** command could not define the partition.

CFG_APIE One or more parameters are incorrect.

CFG_CFEF The VRM call to create the partition failed.

CFG_UNRW An unrecoverable read or write error occurred.

CFG_FOPN An error occurred while opening a file.

Files

/etc/ddi/cpmgr
/etc/filesystems
/etc/system

Related Information

In this book: "attributes" on page 4-20, "master" on page 4-98, and "system" on page 4-139.

The **vrmconfig** command in *AIX Operating System Commands Reference.*

cfgaply

Purpose

Applies configuration information.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg03.h>

int cfgaply (*restart***)**
int *restart***;**

Description

The **cfgaply** subroutine rebuilds the AIX kernel. It runs the **make** command to allow you to rebuild the following files:

- **/usr/sys/cf/conf.c**
- **/usr/sys/cf/conf.o**
- **/usr/sys/specials**
- Kernel library files:
  /usr/sys/lib0
  /usr/sys/lib1
  /usr/sys/lib2

The *restart* parameter indicates whether to restart the system after the subroutine completes. If *restart* is a nonzero value, then the system is restarted after completion.

Before attempting to rebuild the kernel, **cfgaply** creates a backup copy of it named **/unix***date.seq*, where *date* is the Julian date the backup was created, and *seq* is a sequence number, starting with 1. Since kernel images take up storage space, **cfgaply** deletes any backup copies that were saved previously.

The **cfgaply** subroutine creates a shell procedure named **/usr/sys/specials** that contains the **mknod**, **chown**, and **chmod** commands necessary to create the special files (**/dev** files) needed by the new kernel. The **cfgaply** subroutine does *not* run this shell procedure.

Return Value

If the *restart* parameter is nonzero, then the system is restarted and the **cfgaply** subroutine does not return. If *restart* is 0 and **cfgaply** completes successfully, then it returns the value **CFG_SUCC**. If an AIX program fails that **cfgaply** has executed, the return code from that program is returned. If the **cfgaply** subroutine itself fails, then one of the following values is returned:

CFG_ACCS A failure occurred while accessing the **/unix** kernel.

CFG_AOPN The **open** system call failed.

CFG_ABCK A failure occurred while reading the kernel and writing to the backup file.

CFG_ACPF A failure occurred while reading the rebuilt kernel (**/usr/sys/unix.std**) and copying it to **/unix** on the root file system. The previous **/unix** kernel remains intact.

CFG_AMKF The **make** command failed. Error messages from **make** are redirected to the file **/usr/sys/make.out**.

Files

/unix
/unix*date.seq*
/usr/sys/cf/conf.c
/usr/sys/cf/conf.o
/usr/sys/specials
/usr/sys/lib0
/usr/sys/lib1
/usr/sys/lib2
/usr/sys/make.out
/usr/sys/unix.std

Related Information

In this book: "config" on page 6-7.

The **config** and **make** commands in *AIX Operating System Commands Reference.*

cfgcadsz

Purpose

Adds or replaces a stanza in an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg04.h>

int cfgcadsz (*atfile*, *stanza*, *stname*,*after***)**
CFG__SFT **atfile*;
char **stanza*;
char **stname*;
char **after*;

Description

The **cfgcadsz** subroutine adds a new stanza or replaces an existing stanza in an attribute file. (For details about attribute files, see "attributes" on page 4-20.)

The *atfile* parameter points to an open attribute file structure. The *stanza* parameter points to the buffer that contains the stanza to be written. The *stname* parameter points to the name of the stanza to be added to the file.

The *after* parameter points to the name of the stanza after which the new stanza is to be inserted. If this parameter is **NULL**, then the stanza is added to the end of the file.

All information that is repeated in the **default** stanza of the attribute file is removed from the new stanza before it is written to the file.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcadsz** subroutine fails, then the following value is returned:

CFG_ECLS	An error occurred while closing a file.
CFG_EOPN	An error occurred while opening a file.
CFG_SPCE	Memory allocation failed because of insufficient space.
CFG_UNIO	An unrecoverable I/O error occurred during processing.

Related Information

In this book: "cfgadev" on page 3-15, "cfgamni" on page 3-19, "cfgcclsf" on page 3-25, "cfgcdlsz" on page 3-27, "cfgcopsf" on page 3-29, "cfgcrdsz" on page 3-31, and "attributes" on page 4-20.

cfgcclsf

Purpose

Closes an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg04.h>

int cfgcclsf (*atfile***)**
CFG__SFT **atfile*;

Description

The **cfgcclsf** subroutine closes an attribute file. (For details about attribute files, see "attributes" on page 4-20.)

The *atfile* parameter points to an open attribute file structure.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcclsf** subroutine fails, then the following value is returned:

CFG_UNIO Unrecoverable I/O error occurred during processing.

Related Information

In this book: "cfgcadsz" on page 3-23, "cfgcdlsz" on page 3-27, "cfgcopsf" on page 3-29, "cfgcrdsz" on page 3-31, and "attributes" on page 4-20.

cfgcdlsz

Purpose

Deletes a stanza from an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg04.h>

int cfgcdlsz (*atfile*, *stname***)**
CFG__SFT **atfile*;
char **stname*;

Description

The **cfgcdlsz** subroutine deletes a stanza from an attribute file. (For details about attribute files, see "attributes" on page 4-20.)

The *atfile* parameter points to an open attribute file structure. The *stname* parameter points to the name of the stanza to be deleted from the file.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcdlsz** subroutine fails, then one of the following values is returned:

CFG_ECLS	An error occurred while closing a file.
CFG_EOPN	An error occurred while opening a file.
CFG_SPCE	Memory allocation failed because of insufficient space.
CFG_SZBF	The file contains a stanza that is larger than the maximum allowable stanza size.

CFG_SZNF The requested stanza to be deleted was not found in the file.

CFG_UNIO An unrecoverable I/O error occurred during processing.

Related Information

In this book: “cfgcrdsz” on page 3-31, “cfgdmni” on page 3-36, “cfgcadsz” on page 3-23, “cfgcclsf” on page 3-25, “cfgcopsf” on page 3-29, “cfgddev” on page 3-33, and “attributes” on page 4-20.

cfgcopsf

Purpose

Opens an attribute file.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg04.h>

CFG__SFT *cfgcopsf (*path*)
**char ****path*;

Syntax

The **cfgcopsf** subroutine opens an attribute file for update. (For details about attribute files, see "attributes" on page 4-20.)

The *path* parameter points to the full path name of the file to be opened.

The **cfgcopsf** subroutine calls the **fopen** subroutine to open the file for update. If the call to **fopen** is successful, then **cfgcopsf** allocates a **CFG__SFT** structure. This structure contains the file descriptor returned by **fopen**, a pointer to a default stanza buffer for reads, a pointer to an array of indexes in a default stanza buffer, and the full path name of the file that was opened.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the **cfgcopsf** subroutine returns a pointer to an open attribute file structure. If the **cfgcopsf** subroutine fails, it returns a **NULL** pointer.

Related Information

In this book: "cfgcadsz" on page 3-23, "cfgcclsf" on page 3-25, "cfgcdlsz" on page 3-27, "cfgcrdsz" on page 3-31, "fopen, freopen, fdopen" on page 3-168, and "attributes" on page 4-20.

cfgcrdsz

Purpose

Reads an attribute file stanza.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg04.h>

int cfgcrdsz (*atfile*, *stanza*, *nbytes*, *stname***)**
CFG__SFT **atfile*;
char **stanza*;
int *nbytes*;
char **stname*;

Description

The **cfgcrdsz** subroutine reads one stanza from an attribute file. A specific stanza may be requested, or the next stanza in the file can be read. When a stanza is read, any information contained in a **default** stanza preceding it in the file will be added to the information returned in the buffer. (For details about attribute files, see "attributes" on page 4-20.)

The *atfile* parameter points to an open attribute file structure.

The *stanza* parameter points to the buffer into which the stanza will be read.

The *nbytes* parameter is the size in bytes of the buffer pointed to by the *stanza* parameter.

The *stname* parameter points to a string containing the name of the stanza to be read. If this parameter is a **NULL** pointer, then the next stanza in the file is read.

The calling program must have an effective user ID of superuser to access system customization files such as **/etc/master**, **/etc/system**, and **/etc/predefined**.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgcrdsz** subroutine fails, then one of the following values is returned:

CFG_EOF The next stanza was requested, but the end of the file has been reached.

CFG_SZNF The requested stanza was not found in the file.

CFG_SZBF The requested stanza is longer than *nbytes* bytes.

CFG_UNIO Unrecoverable I/O error occurred during processing.

Related Information

In this book: "cfgcadsz" on page 3-23, "cfgcclsf" on page 3-25, "cfgcdlsz" on page 3-27, "cfgcopsf" on page 3-29, and "attributes" on page 4-20.

cfgddev

Purpose

Deletes a device.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg01.h>

int cfgddev (*master*, *system*, *dstname*, *vflag***)**
char **master*, **system*, **dstname*;
int *vflag*;

Description

The **cfgddev** subroutine deletes information about devices and device drivers from the system configuration.

The *master* parameter points to the full path name of the **master** file. The *system* parameter points to the full path name of the **system** file. These files are usually **/etc/master** and **/etc/system**, respectively. The *dstname* parameter points to a string containing the name of the stanza in the *system* file of the device to be deleted.

The *vflag* parameter is either 1 (for "yes") or 0 (for "no"). If the *vflag* parameter is 1, then **cfgddev** executes the **vrmconfig** command with the **-d** *dstname* flag. The **vrmconfig** command then processes the named stanza for driver deletion and produces a shell procedure. The **cfgddev** subroutine then runs this shell procedure to delete the special file (**/dev** file) for the device. If the **vrmconfig** command returns an error, then the device is not deleted.

The **cfgddev** subroutine then gets the VRM device driver stanza associated with the device being deleted. If it contains a **code** keyword, then **cfgddev** searches the *master* file for other VRM device driver stanzas that copy this stanza by specifying its name as the value of a **copy** keyword. If any are found, they are updated so that the first of these stanzas defined in the *master* file contains the **code** keyword, and the other stanzas copy the first stanza.

If the VRM device driver stanza for the device being deleted contains a **copy** keyword, then it is replaced with a **code** keyword whose value is the same as the value of the **code** keyword in the stanza it is copying.

If the device stanza being deleted from the *system* file contains an **admgr** keyword, then its value is the name of the device manager's stanza in the *system* file. The device is deleted from the **vdmgr** keyword value list in the device manager's stanza.

If the device stanza named by the *dstname* parameter contains the **specproc** keyword, then the program specified by the value of this keyword is executed to perform any special processing required when deleting this device. The value of the **specproc** keyword must be the full path name of an executable file. The following arguments are passed to the program using the **argv** mechanism described in "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34. All of them are passed as character strings.

argv[0] The full path name of the special-processing program
argv[1] The full path name of the *master* file
argv[2] The full path name of the *system* file
argv[3] The name of the device stanza
argv[4] The character string "d", indicating deletion.

If the special processing program fails, then the device is still deleted from the system, but some additional steps may be required to clean up the system.

The device stanza associated with the deleted device is then deleted from the *system* file.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgddev** subroutine fails then one of the following values is returned:

CFG_CLSE An error was detected while trying to close a file.

CFG_CPYF A failure occurred while trying to update the VRM driver stanzas that copy the driver stanza of the device being deleted.

CFG_DVND The device could not be deleted from the *system* file.

CFG_DVNF The device to be deleted cannot be found in the *system* file.

CFG_FCOR The *master* or *system* file is set up incorrectly.

CFG_MALF Memory allocation failed because of insufficient space.

CFG_MGRF A failure occurred while updating the device manager's stanza for the device being deleted.

CFG_OPNE	An error was detected while trying to open a file.
CFG_SLPF	Special processing failed. The device is deleted but some additional steps may be required to clean up the system.
CFG_VCFG	The **vrmconfig** command failed.

Files

/etc/specials

Related Information

In this book: "attributes" on page 4-20, "master" on page 4-98, and "system" on page 4-139.

The **vrmconfig** command in *AIX Operating System Commands Reference.*

cfgdmni

Purpose

Deletes a minidisk.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <cfg02.h>

int cfgdmni (*sysstname***,** *fsstname***)**
char ****sysstname*, ****fsstname***;**

Description

The **cfgdmni** subroutine deletes a minidisk from the system. The necessary steps to accomplish this include calling the minidisk manager, executing the **vrmconfig** command, removing the minidisk stanzas from the **/etc/system** and **/etc/filesystems** files, and removing the coprocessor stanza references.

The *sysstname* parameter is a pointer to the name of the stanza in the **/etc/system** file that describes the minidisk that is to be deleted.

The *fsstname* parameter is a pointer to the name of the stanza in the **/etc/filesystems** file if the minidisk is an AIX minidisk. If the minidisk is not an AIX minidisk, then the *fsstname* parameter must be **NULL**.

After the **cfgdmni** subroutine completes successfully, the minidisk is deleted from the system and the minidisk stanza has been deleted from the **/etc/system** file, and, if appropriate, from the **/etc/filesystems** file or from the **/etc/ddi/cpmgr** file.

Return Value

Upon successful completion, the value **CFG_SUCC** is returned. If the **cfgdmni** subroutine fails, then one of the following is returned:

CFG_NSID	The calling process's effective user ID is not superuser.
CFG_VRMF	The **vrmconfig** command could not delete the partition.
CFG_APIE	One or more parameters are incorrect.
CFG_CFEF	The VRM call to delete the partition failed.
CFG_USZF	The stanza specified by the *sysstname* parameter could not be found in the **/etc/system** file.
CFG_UNRW	An unrecoverable read or write error occurred.
CFG_FOPN	An error occurred while opening a file.

Files

/etc/ddi/cpmgr
/etc/filesystems
/etc/system

Related Information

In this book: "attributes" on page 4-20, "master" on page 4-98, and "system" on page 4-139.

The **vrmconfig** command in *AIX Operating System Commands Reference.*

clock

Purpose

Reports CPU time used.

Library

Standard C Library (**libc.a**)

Syntax

long clock ( )

Description

The **clock** subroutine returns the amount of CPU time (in microseconds) used since the first call to **clock**.

The time reported is the sum of the user and system times of the calling process and its terminated child processes for which it has executed a **wait** system call or a **system** subroutine. The nominal resolution of the clock is 16.667 milliseconds if the process is being profiled; otherwise, it is 100 milliseconds. See "monitor" on page 3-248 and "profil" on page 2-99 for information about profiling a process.

Note: The value returned by the **clock** subroutine is defined in microseconds for compatibility with systems that have CPU clocks with much higher resolution. Because of this, the value returned wraps around after accumulating approximately 2147 seconds of CPU time (about 36 minutes).

Related Information

In this book: "times" on page 2-165, "wait" on page 2-182, and "system" on page 3-350.

conv

Purpose

Translates characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;

int NCesc (xp, cp)
NLchar *xp;
char *cp;

int NCtoupper (x)
int x;

int NCtolower (x)
int x;

int _NCtoupper (x)
int x;

int _NCtolower (x)
int x;

int NCtoNLchar (x)
int x;

int NCunesc (cp, xp)
char *cp;
NLchar *xp;

int NCflatchr (x)
int x;
```

Description

The NC*xxxxxxx* subroutines translate all characters, including extended characters, as code points (see "Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System*). The other subroutines translate traditional ASCII characters only.

The **toupper** and the **tolower** subroutines have as domain the range of the **getc** subroutine: from -1 through 255.

If the parameter of the **toupper** subroutine represents a lowercase letter, the result is the corresponding uppercase letter. If the parameter of the **tolôwer** subroutine represents an uppercase letter, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_toupper** and **_tolower** routines are macros that accomplish the same thing as **toupper** and **tolower**, but they have restricted domains and they are faster. **_toupper** requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. **_tolower** requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The value of *x* is in the domain of any legal **NLchar** in a value range from 0 to **NLCHARMAX** inclusive, or a special value of -1 (which represents EOF).

If the parameter of the **NCtoupper** subroutine represents a lowercase letter according to the current collating sequence configuration, the result is the corresponding uppercase letter. If the parameter of the **NLtolower** subroutine represents an uppercase letter according to the current collating sequence configuration, the result is the corresponding lowercase letter. All other values in the domain are returned unchanged.

The **_NCtoupper** and **_NCtolower** routines are macros that accomplish the same thing as **NCtoupper** and **NCtolower**, but have restricted domains and are faster. **_NCtoupper** requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. **_NCtolower** requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Values outside the domain cause undefined results.

The **toascii** subroutine yields the value of its parameter with all bits that are not part of a standard ASCII character turned off. It is intended for compatibility with other systems.

The **NCtoNLchar** subroutine yields the value of its parameter with all bits turned off that are not part of an **NLchar**.

The **NCesc** macro converts the **NLchar** value *xp* into one or more ASCII bytes stored in the character array pointed to by *cp*. If the **NLchar** represents an extended character, it is converted into a printable ASCII escape sequence that uniquely identifies the extended character. **NCesc** returns the number of bytes it wrote. See "display symbols" on page 5-24 for a list that shows the escape sequence for each character.

The inverse conversion is performed by the **NCunesc** macro, translating an ordinary ASCII byte or escape sequence starting at *cp* into a single **NLchar** at *xp*. **NCunesc** returns the number of bytes it read.

The **NCflatchr** subroutine converts its parameter value into the single ASCII byte that most closely resembles the parameter character in appearance. If no ASCII equivalent exists, it converts the parameter value to a ? (question mark).

Related Information

In this book: "ctype" on page 3-49, "getc, fgetc, getchar, getw" on page 3-204, and "display symbols" on page 5-24.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

create_ipc_prof

Purpose

Creates a profile for an IPC queue.

Library

IPC Library (**libipc.a**)

Syntax

#include <drs.h>

int creat_ipc_prof (*queue_name*, *l_key*, *r_key*, *nickname*)

char **queue_name*;
key_t **l_key*, **r_key*;
char **nickname*;

Description

The **create_ipc_prof** subroutine creates a profile for an IPC queue.

The *queue_name* parameter contains the name of the IPC queue. If this value is not specified by the caller, **create_ipc_prof** assigns a queue name and places it in *queue_name*. A *queue_name* supplied by the caller must have valid AIX filename syntax. A *queue_name* supplied by the subroutine has valid AIX filename syntax and is up to 15 characters long, including the trailing **NULL**.

The *l_key* parameter points to the local key for an IPC queue. If this parameter is **NULL**, then **create_ipc_prof** assigns a local key value and places it in *l_key*. If the caller supplies the *l_key*, it should fall between 0x30000 and 0xFFFFF because other ranges are reserved. A value for this parameter supplied by the subroutine will fall in the same range.

The **create_ipc_profile** creates a profile that maps a key (*l_key*) to another key (*r_key*), which can be either local or remote. The *nickname* parameter points to the nickname or node ID, in hexadecimal, of the node where the IPC queue exists. A value of **NULL** for both *r_key* and *nickname* indicates that the queue is on the local node.

If **create_ipc_prof** succeeds in creating a profile, the **dsipc** command is used to update the kernel's copy of the profiles.

Return Value

Upon successful completion, this function returns a 0, and the parameters *queue_name*, *l_key*, *r_key*, and *nickname* contain the values in the created profile. If an error occurs, then **create_ipc_prof** returns a negative value from the following list:

DRS_ACCES	The required access permissions were denied.
DRS_BADLEN	An incorrect parameter was supplied.
DRS_NOREC	No record was found.
DRS_BKEY	The key is out of range.
DRS_NOKEY	No keys are available.
DRS_IO	An input/output error occurred.
DRS_AGAIN	Unable to start pfsmain.
DRS_BADF	An incorrect file descriptor was supplied.
DRS_BADK	An incorrect index key was supplied.
DRS_BDMSF	An incorrect file or table was supplied.
DRS_BOF	The beginning of the file was encountered.
DRS_DEADLK	A deadlock was detected.
DRS_EOF	The end of the file was encountered.
DRS_FAULT	An incorrect address was supplied.
DRS_FBIG	The maximum file size was exceeded.
DRS_IDRM	The identifier was removed.
DRS_INBLCK	The profile database is locked against updates.
DRS_INTENT	The intentions were denied.
DRS_ISDIR	A write to a directory was attempted.
DRS_MFILE	Too many files, tables, or indexes were open.
DRS_NFILE	The file table overflowed.
DRS_NOENT	No file or directory was found.
DRS_NOMEM	No memory is available.
DRS_NOSPC	No space is available on the device.
DRS_NOTDIR	Not a directory.
DRS_NOTIDX	Not an index.

DRS_PANIC	Abnormal termination occurred.
DRS_RCVRY	The file needs recovery.
DRS_RECLEN	The record length is invalid.
DRS_ROFS	The file system to be accessed is read-only.

Related Information

In this book: "msgctl" on page 2-73, "del_ipc_prof" on page 3-64.1, and "find_ipc_prof" on page 3-166.1.

The **dsipc** command in *AIX Operating System Commands Reference.*

AIX Operating System Programming Tools and Interfaces.

crypt, encrypt

Purpose

Encrypts user passwords.

Library

Standard C Library (**libc.a**)

Syntax

char *crypt (*key***,** *salt***)**
char ****key*, ****salt***;**

void encrypt (*block***)**
char ****block*;**

Description

The **crypt** and **encrypt** subroutines encrypt user passwords. They are based on a one-way hashing encryption algorithm with variations intended to frustrate the use of hardware-implemented key searches. These subroutines are provided for compatibility with UNIX system implementations, and no assertion is made about the strength of the algorithm.

The *key* parameter is a user's typed password. The *salt* parameter is a two-character string chosen from the set `[a-zA-Z0-9./]`.

The *salt* parameter is used to perturb the hashing algorithm in one of 4096 different ways, after which the password is used as the key to repeatedly encrypt a constant string. The return value points to the encrypted password. The first two characters of the return value are the string entered in the *salt* parameter.

The **crypt** subroutine uses a character array of length 64 containing only the values `(char) 0` and `(char) 1`. This string is divided into groups of eight characters each, and the low-order bit in each group is ignored. This provides a 56-bit key, which is set into the machine by **crypt**.

The **encrypt** subroutine provides somewhat primitive access to the actual hashing algorithm. The *block* parameter is a 64-character array containing only the values `(char) 0` and `(char) 1`. **encrypt** modifies this array in place, producing a similar array that has been subjected to the hashing algorithm using the key set by **crypt**.

Return Value

The **crypt** subroutine returns a pointer to the encrypted password. The first two characters of it are the same as the *salt* parameter.

Note: The return value points to static data that is overwritten by subsequent calls.

Related Information

In this book: "getpass" on page 3-217 and "passwd" on page 4-112.

The **login** and **passwd** commands in *AIX Operating System Commands Reference.*

ctermid

Purpose

Generates a file name for terminal.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

char *ctermid (*s***)**
char ***s*;**

Description

The **ctermid** subroutine generates the path name of the controlling terminal for the current process and stores it in a string.

If the *s* parameter is a **NULL** pointer, the string is stored in an internal static area and the address is returned. The next call to **ctermid** overwrites the contents of the internal static area.

If the *s* parameter is not a **NULL** pointer, it points to a character array of at least **L_ctermid** elements as defined in the **stdio.h** header file. The path name is placed in this array and the value of *s* is returned.

The difference between the **ctermid** and **ttyname** subroutines is that **ttyname** must be handed a file descriptor and returns the actual name of the terminal associated with that file descriptor, while **ctermid** returns a string (**/dev/tty**) that refers to the terminal if used as a file name. Thus **ttyname** is useful only if the process already has at least one file open to a terminal.

Related Information

In this book: "ttyname, isatty" on page 3-367.

ctime, localtime, gmtime, asctime, tzset

Purpose

Converts date and time to string representation.

Library

Standard C Library (**libc.a**)

Syntax

#include <time.h>

char *ctime (*clock***)**
long ***clock*;**

struct tm *localtime (*clock***)**
long ***clock*;**

struct tm *gmtime (*clock***)**
long ***clock*;**

char *asctime (*tm***)**
struct tm ***tm*;**

void tzset ( )

extern long timezone;
extern int daylight;
extern char *tzname[2];

Description

The **ctime** subroutine converts a time value pointed to by the *clock* parameter, which represents the time in seconds since 00:00:00 Greenwich Mean Time (GMT), January 1, 1970, into a 26-character string in the following form:

```
Sun Sep 16 01:03:52 1973\n\0
```

The width of each field is always the same as shown here.

The **localtime** subroutine converts the long integer pointed to by the *clock* parameter, which contains the time in seconds since 00:00:00 GMT, January 1, 1970, into a **tm** structure. **localtime** adjusts for the time zone and for daylight savings time, if it is in effect.

The **gmtime** subroutine converts the long integer pointed to by the *clock* parameter into a **tm** structure containing the Greenwich Mean Time, which is the time that AIX uses.

The **tm** structure is defined in the **time.h** header file, and it contains the following members:

```
int tm_sec;   /* Seconds (0 - 59) */
int tm_min;   /* Minutes (0 - 59) */
int tm_hour;  /* Hours (0 - 23) */
int tm_mday;  /* Day of month (1 - 31) */
int tm_mon;   /* Month of year (0 - 11) */
int tm_year;  /* Year - 1900 */
int tm_wday;  /* Day of week (Sunday = 0) */
int tm_yday;  /* Day of year (0 - 365) */
int tm_isdst; /* Nonzero = Daylight savings time */
```

The **asctime** subroutine converts a **tm** structure to a 26-character string of the same format as **ctime**.

If the **TZ** environment variable is defined, then its value overrides the default time zone, which is the U.S. Eastern time zone. See "environment" on page 5-47 for the format of the time zone information specified by **TZ**. **TZ** is usually set when the system is started up to the value that is defined in either **/etc/environment** or **/etc/profile**. However, it can also be set by the user as a regular environment variable for performing alternate time zone conversions.

The **tzset** subroutine sets the **timezone**, **daylight**, and **tzname** external variables to reflect the setting of **TZ**. **tzset** is called by **ctime** and **localtime**, and it can also be called explicitly by an application program.

The **timezone** external variable contains the difference, in seconds, between GMT and local standard time. For example, **timezone** is 5 × 60 × 60 for U.S. Eastern Standard Time.

The **daylight** external variable is non-zero when a daylight savings time conversion should be applied. By default, this conversion follows the standard U.S. conventions; other conventions can be specified. The default conversion algorithm adjusts for the peculiarities of U.S. daylight savings time in 1974 and 1975. See "environment" on page 5-47 for information about specifying alternate daylight savings time conventions.

The **tzname** external variable contains the name of the standard time zone (**tzname[0]**) and of the time zone when daylight savings time is in effect (**tzname[1]**). For example:

```
char *tzname[2] = {"EST", "EDT"};
```

The **time.h** header file contains declarations of all these subroutines, externals, and the **tm** structure.

Warning: The return values point to static data that is overwritten by each call.

Related Information

In this book: "time" on page 2-164, "getenv, NLgetenv" on page 3-208, "NLstrtime" on page 3-288, "NLtmtime" on page 3-291, "profile" on page 4-127, and "environment" on page 5-47.

ctype

Purpose

Classifies characters.

Library

Standard C Library (**libc.a**)

Syntax

#include <ctype.h>

int isalpha (*c***)**
int *c***;**

int isupper (*c***)**
int *c***;**

int islower (*c***)**
int *c***;**

int isdigit (*c***)**
int *c***;**

int isxdigit (*c***)**
int *c***;**

int isalnum (*c***)**
int *c***;**

int isspace (*c***)**
int *c***;**

int ispunct (*c***)**
int *c***;**

int isprint (*c***)**
int *c***;**

int isgraph (*c***)**
int *c***;**

int iscntrl (*c***)**
int *c***;**

int isascii (*c***)**
int *c***;**

Description

The **ctype** macros classify character-coded integer values by table look-up. Each of these macros returns a nonzero value for "true" and 0 for "false."

The **isascii** macro is defined for all integer values. The other macros return a meaningful value only if **isascii** returns "true" for the same *c* value, or if *c* is **EOF**. (See "standard i/o library" on page 3-342 for information about the value **EOF**.)

The following list shows the set of values for which each macro returns a nonzero ("true") value:

isalpha *c* is a letter.

isupper *c* is an uppercase letter.

islower *c* is a lowercase letter.

isdigit *c* is a digit in the range [0-9].

isxdigit *c* is a hexadecimal digit in the range [0-9], [A-F] or [a-f].

isalnum *c* is alphanumeric (a letter or a digit).

isspace *c* is a space, tab, carriage return, new-line, vertical tab, or form-feed character.

ispunct *c* is a punctuation character (neither a control character nor alphanumeric).

isprint *c* is a printing character, ASCII space (040 or 0x20) through ~ (0176 or 0x7E).

isgraph *c* is a printing character, like **isprint** but, unlike **isprint**, **isgraph** returns false (0) for the space character.

iscntrl *c* is an ASCII DEL character (0177 or 0x7F) or an ordinary control character (less than 040 or 0x20).

isascii *c* is an ASCII character whose value is in the range 0−0177 (0−0x7F), inclusive.

Related Information

In this book: "NCctype" on page 3-270, "ascii" on page 5-3, and "data stream" on page 5-5.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

curses

Purpose

Controls cursor movement and windowing.

Library

Curses Library (**libcurses.a**)

Syntax

#include <curses.h>
#include <term.h>

Description

Note: The **curses** package of subroutines is included here only for compatibility with existing programs. For information about the enhanced screen-handling subroutine library, see "extended curses library" on page 3-131.

The **curses** subroutine package updates the screen with reasonable optimization. The **term.h** header file is only needed if **terminfo** level routines are needed (see "Terminfo Level Subroutines" on page 3-57).

In order to initialize the routines, the routine **initscr** must be called before any of the other routines that deal with windows and screens are used. The routine **endwin** should be called before exiting. To get character-at-a-time input without echoing, call the **nonl**, **cbreak**, and **noecho** routines. Most interactive, screen-oriented programs require the character-at-a-time input without echoing.

The full **curses** interface permits manipulation of data structures called ***windows***, which can be thought of as two-dimensional arrays of characters representing all or part of a screen. A default window called **stdscr** is supplied, and others can be created with the **newwin** routine. Windows are referred to by variables declared **WINDOW ***. The type **WINDOW** is defined in **curses.h** to be a C structure. These data structures are manipulated with the routines described following, among which the most basic are **move** and **addch**. Then the **refresh** routine is called, telling the routines to make the screen look like **stdscr**. More general versions of these routines are included with names beginning with **w** allowing you to specify a window. The routines not beginning with a **w** affect **stdscr**.

Minicurses is a subset of **curses** that does not allow manipulation of more than one window. To invoke this subset, use **-DMINICURSES** as a **cc** option. This level is smaller and faster than the full **curses.**

If the environment variable **TERMINFO** is defined, any program using **curses** checks for a local terminal definition before checking in **/usr/lib/terminfo**. For example, **TERM** is set to **vt100**, then normally, the compiled file is found in **/usr/lib/terminfo/v/vt100**. (The directory name **v** is copied from the first letter of **vt100** to avoid creating huge directories.) If, for example, **TERMINFO** is set to **/usr/mark/myterms**, **curses** first checks **/usr/mark/myterms/v/vt100**. If this file does not exist, **curses** then checks **/usr/lib/terminfo/v/vt100**. This is useful for developing experimental definitions or when write permission in **/usr/lib/terminfo** is not available.

Note: The plotting library, **plot** and the curses library, **curses** both use the names **erase** and **move**. The **curses** versions are macros. If you need both libraries, put the **plot** code in a different source file than the **curses** code, or include the following statements in the **plot** code:

```
#undef move()
#undef erase()
```

Routines

The routines listed here can be called when using the full **curses**. Those marked with an asterisk can be called when using **minicurses**.

addch(*ch*)*	Add a character to **stdscr** (like **putchar**), wrapping to the next line at the end of a line.
waddch(*win, ch*)	Add the character *ch* to *win*
mvwaddch(*win*, *y*, *x*, *ch*)	Move (*y*, *x*) then add the character *ch* to *win*
addstr(*str*)*	Call **addch** with each character in *str*
mvaddstr(*y, x, str*)	Move (*y*, *x*) then add *str*
waddstr(*win, str*)	Add the string *str* to *win*
mvwaddstr(*win, y, x, str*)	Move (*y*, *x*) then add the string *str* to *win*
attroff(*attrs*)*	Turn off the attributes named in *attrs*
attron(*attrs*)*	Turn on the attributes named in *attrs*
attrset(*attrs*)*	Set current attributes to those specified in *attrs*
baudrate ( )*	Set current terminal speed
beep ( )*	Sound beep on terminal

box(*win, vert, hor*)	Draw a box around edges of *win*. The *vert* and *hor* parameters are the characters to use for vertical and horizontal edges of the box.
cbreak ( )*	Set cbreak mode
nocbreak ( )*	Unset cbreak mode
clear ( )	Clear **stdscr**
clearok(*win, bf*)	Clear screen before next redraw of *win*
clrtobot ( )	Clear to bottom of **stdscr**
clrtoeol ( )	Clear to end of line on **stdscr**
delay_output(*ms*)*	Insert *ms* millisecond pause in output
nodelay(*win, bf*)	Enable **nodelay** input mode through **getch**
delch ( )	Delete a character
deleteln ( )	Delete a line
delwin(*win*)	Delete window *win*
doupdate ( )	Update screen from all **wnoutrefresh**
echo ( )*	Set echo mode
noecho ( )*	Unset echo mode
endwin ( )*	End window modes
erase ( )	Erase **stdscr**
erasechar ( )	Return user's erase character
fixterm ( )	Restore terminal to *in curses* state
flash ( )	Flash screen or beep
flushinp ( )*	Throw away any type-ahead
getch ( )*	Get a character from **tty**
getstr(*str*)	Get a string through **stdscr**
gettmode ( )	Establish current **tty** modes
getyx(*win*, *y*, *x*)	Get (*y*, *x*) coordinates
has_ic ( )	Has value of `true` if terminal can do insert character
has_il ( )	Has value of `true` if terminal can do insert line
idlok(*win*, *bf*)*	Use terminal's insert/delete line if *bf*!=0

inch ( )	Get character at current (*y*, *x*) coordinates
initscr ( )*	Initialize screens
insch(*c*)	Insert a character
insertln ( )	Insert a line
intrflush(*win*, *bf*)	Interrupt flush output if *bf* is true
keypad(*win*, *bf*)	Enable keypad input
killchar ( )	Return current user's **kill** character
leaveok(*win*, *flag*)	Permit cursor to be left anywhere after refresh if *flag*! = 0 for *win*; otherwise cursor must be left at current position
longname ( )	Return verbose name of terminal
meta(*win*, *flag*)*	Allow metacharacters on input if *flag*! = 0
move(*y*, *x*, *ch*)*	Move to (*y*, *x*) on **stdscr**
mvaddch(*y*, *x*, *ch*)	Move (*y*, *x*) then add *ch*
mvcur(*oldrow*, *oldcol*, *newrow*, *newcol*)	Move cursor from current position to another position
mvdelch(*y*, *x*)	Move (*y*, *x*) then delete a character
mvgetch(*y*, *x*)	Move (*y*, *x*) then get a character from **tty**
mvgetstr(*y*, *x*, *str*)	Move (*y*, *x*) then get a string through **stdscr**
mvinch(*y*, *x*)	Move (*y*, *x*) then get the character at current (*y*, *x*) coordinates
mvinsch(*y*, *x*, *c*)	Move (*y*, *x*) then insert the character *c*
mvprintw(*y*, *x*, *fmt*, *args*)	Move (*y*, *x*) then get print on **stdscr**
mvscanw(*y*, *x*, *fmt*, *args*)	Move (*y*, *x*) then scan through **stdscr**
mvwdelch(*win*, *y*, *x*)	Move (*y*, *x*) then delete a character from *win*
mvwgetch(*win*, *y*, *x*)	Move (*y*, *x*) then get a character through *win*
mvwgetstr(*win*, *y*, *x*, *str*)	Move (*y*, *x*) then get a string through *win*
mvwin(*win*, *by*, *bx*)	Move *win* so that the upper left-hand corner is located at (*y*, *x*)
mvwinch(*win*, *y*, *x*)	Move (*y*, *x*) then get the character at current (*y*, *x*) in *win*
mvwinsch(*win*, *y*, *x*, *c*)	Move (*y*, *x*) then insert the character *c* into *win*

mvwprintw(*win*, *y*, *x*, *fmt*, *args***)**	Move (*y*, *x*) then **printf** on **stdscr**
mvwscanw(*win*, *y*, *x*, *fmt*, *args***)**	Move (*y*, *x*) then **scanf** through **stdscr**
newpad(*nlines*, *ncols***)**	Create a new pad with given dimensions
newterm(*type*, *fd***)**	Set up new terminal of given type to output on *fd*
newwin(*lines*, *cols*, *begin_y*, *begin_x***)**	Create a new window
nl ( )*	Set newline mapping
nonl ( )*	Unset newline mapping
overlay(*win1*, *win2***)**	Overlay *win1* on *win2*
overwrite(*win1*, *win2***)**	Overwrite *win1* on top of *win2*
printw(*fmt*, *arg1*, *arg2*, ...**)**	Print on **stdscr**
raw ( )*	Set raw mode
refresh ( )*	Make current screen look like **stdscr**
prefresh(*pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol***)**	Refresh from *pad* starting with given upper left corner of pad with output to given portion of screen
pnoutrefresh(*pad*, *pminrow*, *pmincol*, *sminrow*, *smincol*, *smaxrow*, *smaxcol***)**	Refresh like **prefresh**, but with no output until **doupdate** is called
noraw ( )*	Unset **raw** mode
resetterm ( )*	Set **tty** modes to *out of curses* state
resetty ( )*	Reset **tty** flags to stored value
saveterm ( )*	Save current modes as *in curses* state
savetty ( )*	Store current **tty** flags
scanw(*fmt*, *arg1*, *arg2*, ...**)**	Scanf through **stdscr**
scroll(*win***)**	Scroll *win* one line
scrollok(*win*, *flag***)**	Allow terminal to scroll if *flag*! = 0
set_term(*new***)**	Enable talk to terminal *new*
setscrreg(*t*, *b***)**	Set user scrolling region to lines *t* through *b*

setterm(*type***)**	Establish terminal with a give type
standend ( )*	Clear standout mode attribute
standout ( )*	Set standout mode attribute
subwin(*win*, *lines*, *cols*, *begin_y*, *begin_x***)**	Create a subwindow
touchwin(*win***)**	Change all of *win*
traceoff ( )	Turn off debugging trace output
traceon ( )	Turn on debugging trace output
typeahead(*fd***)**	Check file descriptor *fd* to check type-ahead
unctrl(*ch***)***	Use printable version of *ch*
wattroff(*win*, *attrs***)**	Turn off *attrs* in *win*
wattron(*win*, *attrs***)**	Turn on *attrs* in *win*
wattrset(*win*, *attrs***)**	Set attributes in *win* to *attrs*
wclear(*win***)**	Clear *win*
wclrtobot(*win***)**	Clear to bottom of *win*
wclrtoeol(*win***)**	Clear to end of line on *win*
wdelch(*win*, *c***)**	Delete the character *c* from *win*
wdeleteln(*win***)**	Delete line from *win*
werase(*win***)**	Erase *win*
wgetch(*win***)**	Get a character through *win*
wgetstr(*win*, *str***)**	Get the string *str* through *win*
winch(*win***)**	Get the character at current (*y*, *x*) in *win*
winsch(*win*, *c***)**	Insert the character *c* into *win*
winsertln(*win***)**	Insert line into *win*
wmove(*win*, *y*, *x***)**	Set current (*y*, *x*) coordinates on *win*
wnoutrefresh(*win***)**	Refresh but no screen output
wprintw(*win*, *fmt*, *arg1*, *arg2*, . . . **)**	**printf** on *win*
wrefresh(*win***)**	Make screen look like *win*

wscanw(*win*, *fmt*, *arg1*, *arg2*, . . .)
scanf through *win*

wsetscrreg(*win*, *t*, *b*) Set scrolling region of *win*

wstandend(*win*) Clear standout attribute in *win*

wstandout(*win*) Set standout attribute in *win*

Terminfo Level Subroutines

These routines should be called by programs that have to deal directly with the **terminfo** data base. Due to the low level of this interface, its use is discouraged. The header files **curses.h** and **term.h** should be included (in that order) to get the definitions for these strings, numbers, and flags. You should call **setupterm** before using any of the other **terminfo** subroutines. This defines the set of terminal-dependent variables defined in the **terminfo** file.

If the program needs only one terminal, you can specify the **-DSINGLE** flag to the C compiler. This results in static references instead of dynamic references to capabilities. The result is smaller code, but only one terminal can be used at a time for the program.

Capabilities with a Boolean value have the value 1 if the capability is present and 0 if it is not. Numeric capabilities have a value of -1 if the capability is missing and a value of 0 or greater if it is present. String capabilities have a **NULL** value if the capability is missing and otherwise have type **char *** and point to a character string that contains the capability. Special character codes that use the backslash and circumflex characters (\ and ^) are transformed into the appropriate ASCII characters. Padding information of the form $<*time*>, and parameter information beginning with % (percent) are left uninterpreted. The **tputs** routine interprets padding information and **tparm** interprets parameter information.

All **terminfo** strings (including the output of **tparm**) should be printed with **tputs** or **putp**. Before exiting, **reset_shell_mode** should be called to restore the **tty** modes. Programs desiring shell escapes can call **reset_shell_mode** before the shell is called and **reset_prog_mode** after returning from the shell.

delay_output (*ms*)
Sets the output delay, in milliseconds.

def_prog_mode
Saves the current terminal mode as program mode, in **cur_term->Nttyb**.

def_shell_mode
Saves the shell mode as normal mode, in **cur_term->Ottyb**. **def_shell_mode** is called automatically by **setupterm**.

putp(*str*)
Calls **tputs**(*str*, **1**, **putchar**).

reset_prog_mode
Puts the terminal into program mode.

reset_shell_mode
Puts the terminal into shell mode. All programs must call **reset_shell_mode** before they exit. The higher-level routine **endwin** automatically does this.

setupterm(*term*, *fd*, *rc***)**
Reads in the data base. *term* is a character string that specifies the terminal name. If *term* is 0, then the value of the **TERM** environment variable is used. One of the following status values is stored into the integer pointed to by *rc*:

1 Successful completion
0 No such terminal
-1 An error occurred while locating the **terminfo** database.

If the *rc* parameter is 0, then no status value is returned, and an error causes **setupterm** to print an error message and exit, rather than return. *fd* is the file descriptor of the terminal being used for output. **setupterm** calls **termdef** to determine the number of lines and columns on the display. If **termdef** cannot supply this information, then **setupterm** uses the values in the **terminfo** data base. The simplest call is **setupterm(0, 1, 0)**, which uses all the defaults.

After the call to **setupterm**, the global variable **cur_term** is set to point to the current structure of terminal capabilities. It is possible for a program to use more than one terminal at a time by calling **setupterm** for each terminal and saving and restoring **cur_term**.

The **setupterm** subroutine also initializes the global variable **ttytype** as an array of characters to the value of the list of names for the terminal. The list comes from the beginning of the **terminfo** description.

tparm(*str*, *p1*, *p2*, . . . *p9***)**
Instantiates the string *str* with parameters p_i. The character string returned has the given parameters applied.

tputs(*str*, *affcnt*, *putc***)**
Applies padding information to string *str*. *affcnt* is the number of lines affected, or 1 if not applicable. *putc* is a **putchar**-like routine to which the characters are passed one at a time.

Some strings are of a form like $<20>, which is an instruction to pad for 20 milliseconds.

vidputs(*attrs*, *putc***)**
Outputs the string to put terminal in video attribute mode *attrs*. Characters are passed to the **putchar**-like routine *putc*. The *attrs* are defined in **<curses.h>**. The previous mode is retained by this routine.

vidattr(*attrs***)**
Like **vidputs**, but outputs through **putchar**.

Termcap Compatibility Routines

These routines are included for compatibility with programs that require **termcap**. Their parameters are the same as for **termcap**, and they are emulated using the **terminfo** data base.

tgetent(*bp*, *name***)**
Looks up the **termcap** entry for *name*. *bp* and *name* are strings. *name* is a terminal name; *bp* is ignored. Calls **setupterm**.

tgetflag(*id***)**
Returns the Boolean entry for *id*. *id* is a 2-character string that contains a **termcap** identifier.

tgetnum(*id***)**
Returns the numeric entry for *id*. *id* is a 2-character string that contains a **termcap** identifier.

tgetstr(*id*, *area***)**
Returns the string entry for *id*. *id* is a 2-character string that contains a **termcap** identifier. The *area* parameter is ignored.

tgoto(*cap*, *col*, *row***)**
Applies parameters to the given *cap*. Calls **tparm**.

tputs(*cap*, *affcnt*, *fn***)**
Applies padding to *cap* calling *fn* as **putchar**.

Attributes

The following video attributes can be passed to the routines **attron**, **attroff**, and **attrset**.

A_STANDOUT	The terminal's best highlighting mode
A_UNDERLINE	Underlined
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_INVIS	Invisible (blanked or zero-intensity)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set
A_NORMAL	Normal attributes

Function Keys

The following function keys might be returned by **getch** if **keypad** has been enabled. Note that not all of these are currently supported due to lack of definitions in **terminfo**, or due to the terminal not transmitting a unique code when the key is pressed.

KEY_BREAK	Break key (unreliable)
KEY_DOWN	Down-arrow key
KEY_UP	Up-arrow key
KEY_LEFT	Left-arrow key
KEY_RIGHT	Right-arrow key
KEY_HOME	Home key
KEY_BACKSPACE	Backspace (unreliable)
KEY_F(*n*)	Function key F*n*, where *n* is an integer from 0 to 63
KEY_DL	Delete line
KEY_IL	Insert line
KEY_DC	Delete character
KEY_IC	Insert character or enter insert mode
KEY_EIC	Exit insert character mode
KEY_CLEAR	Clear screen
KEY_EOS	Clear to end of screen
KEY_EOL	Clear to end of line
KEY_SF	Scroll 1 line forward
KEY_SR	Scroll 1 line backwards (reverse)
KEY_NPAGE	Next page
KEY_PPAGE	Previous page
KEY_STAB	Set tab
KEY_CTAB	Clear tab
KEY_CATAB	Clear all tabs
KEY_ENTER	Enter or send (unreliable)
KEY_SRESET	Soft (partial) reset (unreliable)
KEY_RESET	Reset or hard reset (unreliable)
KEY_PRINT	Print or copy
KEY_LL	Home down or bottom (lower left)
KEY_A1	Upper left key of keypad
KEY_A3	Upper right key of keypad
KEY_B2	Center key of keypad
KEY_C1	Lower left key of keypad
KEY_C3	Lower right key of keypad

Related Information

In this book: "extended curses library" on page 3-131, "termdef" on page 3-352, and "terminfo" on page 4-148.

cuserid

Purpose

Gets the alphanumeric user name associated with the current process.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

char *cuserid (*s***)**
char ****s*;**

Description

The **cuserid** subroutine generates a character string representing the user name of the owner of the current process.

If the *s* parameter is a **NULL** pointer, then the character string is stored into an internal static area, the address of which is returned.

If the *s* parameter is not a **NULL** pointer, then the character string is stored into the array pointed to by the *s* parameter. This array must contain at least **L_cuserid** characters. **L_cuserid** is a constant defined in the **stdio.h** header file.

If the user name cannot be found, the **cuserid** subroutine returns a **NULL** pointer; if the *s* parameter is not a **NULL** pointer, then a null character ('\0') is stored into *s*[0].

Related Information

In this book: "getlogin" on page 3-212, "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-219, and "standard i/o library" on page 3-342.

dbm

Purpose

Performs data base operations.

Library

Database Library (**libdbm.a**)

Syntax

```
int dbminit (file)
char *file;

datum fetch (key)
datum key;

int store (key, content)
datum key, content;

int delete (key)
datum key;
```

```
datum firstkey ( )

datum nextkey (key)
datum key;

typedef struct
{
    char   *dptr;
    int    dsize;
} datum;
```

Description

The **dbm** subroutines maintain a data base of *key-content* pairs. These subroutines can handle very large data bases and access keyed items in one or two file-system accesses.

The *key* parameter is a pointer to data specified by the *content* parameter. The sum of the sizes of the *key-content* pairs must not exceed the internal block size of 512 bytes. All *key-content* pairs that hash together must fit on a single block. The **store** subroutine returns an error if a disk block fills with inseparable data.

The *key* and the *content* parameters are described by the **typedef datum** structure. The **datum** structure sets up a string of bytes. The length of the string is specified by the **dsize** field. The string is pointed to by the **dptr** field. The **dptr** pointers that are returned by these subroutines point to static storage that changes with subsequent calls. You can use binary data or normal ASCII strings.

The data base is stored in two files. One file is a directory that contains a bit map and is suffixed with **.dir**. The second file contains all data and is suffixed with **.pag**. The **.pag** file contains holes that increases its apparent size to about four times its actual size. You cannot copy a **.pag** file using the standard utilities such as **cp** and **cat** without first filling these holes.

Before you can access a data base, you must open the data base with the **dbminit** subroutine. The *file*, **.dir**, and **.pag** files must already exist before you call the **dbminit** subroutine. You can create an empty data base by creating zero-length **.dir** and **.pag** files.

After the data base is opened with the **dbminit** subroutine, you can use the **fetch** subroutine to access the data that is is pointed to by the *key* parameter. You can use the **store** subroutine to write the data specified by the *content* parameter to a file and to specify the key to be used to access that data with the *key* parameter.

The **delete** subroutine removes the key specified by the *key* parameter and the data to which that key points. The **delete** subroutine does not actually reclaim the file space, but it does make it available for reuse.

The **firstkey** and **nextkey** subroutines make a linear pass through all of the keys in a data base. The **firstkey** subroutine returns the first key in the data base. The **nextkey** subroutine returns the next key in the data base. The following code makes a linear pass through a data base:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
   {
    . . .
   }
```

The order of keys that are presented to **firstkey** and **nextkey** depend on the hashing function.

Return Value

All of the **dbm** subroutines that return an *int* value return 0 upon successful completion, and they return a negative value if an error occurs. Subroutines that return a **datum** value indicate an error by setting the **dptr** field to **NULL.**

del_ipc_prof

Purpose

Deletes an IPC queue profile.

Library

IPC Library (**libipc.a**)

Syntax

#include <drs.h>

int del_ipc_prof (*queue_name*, *l_key*, *r_key*, *nickname*)

char **queue_name*;
key_t **l_key*, **r_key*;
char **nickname*;

Description

The **del_ipc_prof** subroutine deletes an IPC queue profile at the local node.

The *queue_name* parameter contains the name of an IPC queue. The *l_key* parameter points to the local key for an IPC queue. You must specify one or both of these values. The **del_ipc_prof** subroutine fails if both *queue_name* and *l_key* are **NULL**.

The *r_key* is a pointer from the local node to the IPC profile for a queue at a remote node. The *nickname* parameter points to the nickname or node ID, in hexadecimal, of the node where the IPC queue exists. A value of **NULL** indicates that the queue is on the local node.

The application does not supply values for the *r_key* and *nickname* parameters. The **del_ipc_prof** subroutine assigns values to these parameters when it returns. The application, however, must ensure that enough space is allocated to hold the return values.

If **del_ipc_prof** successfully deletes the requested profile, the **dsipc** command is used to update the kernel's copy of the profiles.

Return Value

Upon successful completion, the function returns a 0, and *queue_name*, *l_key*, *r_key*, and *nickname* contain the values from the deleted profile. If an error occurs, **del_ipc_prof** returns a negative value from the following list:

DRS_ACCES	The required access permissions were denied.
DRS_BADLEN	An incorrect parameter was supplied.
DRS_NOREC	No record was found.
DRS_IO	An input/output error occurred.
DRS_AGAIN	Unable to start pfsmain.
DRS_BADF	An incorrect file descriptor was supplied.
DRS_BADK	An incorrect index key was supplied.
DRS_BDMSF	An incorrect file or table was supplied.
DRS_BOF	The beginning of the file was encountered.
DRS_DEADLK	A deadlock was detected.
DRS_EOF	The end of the file was encountered.
DRS_FAULT	An incorrect address was supplied.
DRS_FBIG	The maximum file size was exceeded.
DRS_IDRM	Identifier removed.
DRS_INBLCK	The profile database is locked against updates.
DRS_INTENT	Intentions denied.
DRS_ISDIR	A write to a directory was attempted.
DRS_LOCKPM	Lock permission denied.
DRS_MFILE	Too many files, tables, or indexes were open.
DRS_NFILE	The file table overflowed.
DRS_NOENT	No file or directory was found.
DRS_NOMEM	No memory is available.
DRS_NOSPC	No space is available on the device.
DRS_NOTDIR	Not a directory.
DRS_NOTIDX	Not an index.
DRS_PANIC	Abnormal termination occurred.

DRS_RCVRY	File needs recovery.
DRS_RECLEN	Record length is invalid.
DRS_ROFS	The file system to be accessed is read-only.

Related Information

In this book: "msgctl" on page 2-73, "create_ipc_prof" on page 3-40.2, and "find_ipc_prof" on page 3-166.1.

The **dsipc** command in *AIX Operating System Commands Reference.*

AIX Operating System Programming Tools and Interfaces.

DOS services library

Purpose

Provides access to DOS files and diskettes.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

Description

The DOS Services subroutines provide a programming environment for applications that utilize DOS Services. DOS Services is an AIX Operating System shell that interacts with the system user like DOS and provides access to both AIX and DOS file systems. The **dos** command starts this shell. (The **dos** command is discussed in *AIX Operating System Commands Reference.*)

The DOS Services library provides access to DOS file systems on fixed disks and on diskettes in addition to AIX file systems. The access is transparent; that is, applications do not need to know which type of file system provides the files.

Applications intended to be run under DOS Services are actually AIX applications. While the user interface to DOS Services is as similar to DOS as possible, the applications programming interface follows the conventions of AIX and AIX system calls wherever possible. Many AIX applications can be converted to use the DOS Services library with few modifications.

The DOS Services subroutines require that your programs include the **dos.h** header file. When an error occurs, the DOS Services subroutines set the global variable **doserrno** to indicate the error, resembling the error reporting performed by system calls. If you want your program to check **doserrno**, then you must also include the **doserrno.h** header file. For detailed information about header files, see "Header Files" on page vii.

An application program receives parameters in the standard *argc*, *argv*, *envp* format used to pass parameters to ordinary AIX processes. (See "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34 for details about this parameter-passing convention.) The application can use most of the services provided by AIX, but must use the DOS Services subroutines for file access to ensure compatibility with DOS file systems.

Any application can use the **exec** system call and the **dosexecve** subroutine to invoke another AIX program, including another DOS Services application program. There is no way for the invoked program to tell which program invoked it other than by the content of the parameters or the environment. The **exec** system call and the **dosexecve** subroutine do not process DOS Services path information. The DOS path information must be processed by the application program, using the **DOS_PATH** environment variable.

A **.BAT** file cannot be directly invoked by **exec** or **dosexecve**. To execute a **.BAT** file, an application program must run the **dos** command with the appropriate flags and parameters. For example, the following call runs the batch file `hello.bat`:

```
execl ("/usr/bin/dos", "dos", "-a", "-c", "hello.bat", 0);
```

See the **dos** command in *AIX Operating System Commands Reference* for details about the flags and their meanings.

The DOS Services library performs transparent translation of textual data between DOS ASCII and AIX ASCII formats. This translation takes place for a given file if the **DO_ASCII** bit is set set when the file is opened with the **dosopen** subroutine. The application program operates on the data in AIX ASCII format whether the file is located on an AIX file system or on a DOS file system. See "dosread" on page 3-98 and "doswrite" on page 3-116 for more details about the translation performed.

The DOS Services library provides no direct support for interaction with an attached coprocessor. Access to a file system is mediated by the VRM, which prevents the sharing of a file system between the coprocessor and AIX.

The DOS Services library supports the DOS file systems in both diskette and fixed disk formats. It uses the content of the device rather than the device itself to determine the format of the file system. Therefore, it is possible to copy a diskette to a fixed disk using the **cp** command, and to access the diskette data from the fixed disk.

The DOS Services library supports multiplexed disk drives. A ***multiplexed*** drive is a single physical drive that is configured as several logical drives (such as drives **A:** and **B:**). As one or the other of these is accessed, the DOS Services system prompts the user to insert the appropriate diskette.

The DOS Services library provides recovery from diskette I/O errors in the form of `Abort, Retry, Ignore` messages.

The DOS Services library maps DOS file attributes into AIX file modes whenever possible so that the application programmer need think only in terms of AIX file modes. The directory, read-only, and hidden attributes map to corresponding facilities in AIX. The system, volume, and archive attributes are not directly supported, but are recognized by the DOS commands that need to use them. The **dosstat** and **dosfstat** subroutines provide access to the attributes of both DOS and AIX files.

If both a parent and a child process use DOS Services subroutines, then the parent must call **dosunopen** before starting the child process, and it must call **dosreopen** after the child finishes. This synchronizes the information shared by the two processes.

Standard header information required for many of the DOS Services library routines is defined in the file **dos.h**.

DOS Services library routines return diagnostic codes like the AIX system calls. Subroutines return a value of -1 or **NULL** in case of an error, and the variable **doserrno** is set to indicate the error. The file **doserrno.h** contains definitions of each possible DOS diagnostic code. The majority of these codes conform to AIX diagnostic codes.

Device Names

DOS emulation requires binding DOS devices to AIX files. Device names in the DOS environment are mapped to AIX files according to definitions found in the environment at the time the **dosinit** subroutine is first invoked in a process family. Generally, this will be performed by the **dos** command.

Device	Environment Variable and Default Setting
NUL:	DOS_NUL=/dev/null
CON:	DOS_CON=/dev/tty
COM1:	DOS_COM1=/dev/tty0
COM2:	DOS_COM2=/dev/tty1
AUX:	DOS_AUX=/dev/tty0
LP0:	DOS_LP0=/dev/lp0
LP1:	DOS_LP1=/dev/lp1
LP2:	DOS_LP2=/dev/lp2
.	.
.	.
.	.
LP7:	DOS_LP7=/dev/lp7
A:	DOS_A=/dev/fd0
B:	DOS_B=/dev/fd0
C:	DOS_C=$HOME
D:	DOS_D=/

A DOS disk drive name can be bound to an AIX directory, file, or device formatted as an AIX or a DOS file system. Typically, this is **/dev/fd***n* or **/dev/vd***n*. Any uppercase alphabetic character can be used for a DOS disk name.

A DOS nondisk device can be bound to an AIX file or device or to a program. Only the names listed in the preceding table can be used as nondisk devices. If the first character of the value of the AIX path name bound to a DOS nondisk device is a | (vertical bar), the associated device will be a pipe into the shell command given by the rest of the symbol value. If it is not a vertical bar, the value will be interpreted as an AIX file name.

The **dosinit** subroutine creates a configuration table which is propagated to subordinate processes. The environment is not inspected after this table is initialized. Files and devices are not actually opened until they are accessed.

File Naming

A DOS file name has the following format:

[*d*:][*path*]*filename*[.*ext*]

DOS file names are converted to AIX file names as follows:

The characters `A-Z 0-9 $ & # @ ! % ' ` - _ ^ ~`

When file name specifications refer to DOS file systems, lowercase characters in file name specifications are converted to uppercase by the DOS Services subroutines. No translation is made when file names refer to AIX file systems.

d:

The drive name can be any single letter followed by a colon. The DOS Services library translates it to uppercase.

path

The directory path is of the form:

[\][*dirname*][\dirname . . .]

If the file is on an AIX file system, then each directory level is translated to an AIX directory level with the same name.

filename[.*ext*]

A DOS file name consists of a ***filename*** of one to eight characters that can be followed by an extension. The ***extension***, if present, consists of a **.** (period) and up to three characters. AIX file names are 1 to 14 characters long, including the extension. The extension can be from 1 to 13 characters long, including the period. Incompatibilities may arise when copying files from AIX file systems to DOS file systems.

Diagnostics

When a DOS Services subroutine encounters an error, it returns the value -1 and sets the global variable **doserrno** to a code that identifies the error. This scheme resembles the one used by AIX system calls.

All of the possible error codes are listed in the **doserrno.h** header file. For your convenience, they are also listed here:

DE_FNAME (-2)	Syntax error in file name
DE_NOMNT (-3)	AIX file system is not mounted
DE_UNOPEN (-4)	File unopened and not reopened
DE_EXDOS (-5)	Attempt to execute a DOS file
DE_RFULL (-6)	DOS root directory is full
DE_ROOT (-7)	Can not modify DOS root directory
DE_BADMNT (-8)	Bad header or FAT for DOS file system

DE_NEMPTY (-9)	Directory is not empty
DE_INIT (-10)	**dosinit** configuration error
DE_ENVT (-11)	Environment file error

In addition, **doserrno** may be set to any of the values set by the AIX system calls. These values are redefined in the **doserrno.h** header file with the prefix **DE_** added.

dosassign

Purpose

Assigns one DOS Services drive to another.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosassign (*drive*, *todrive***)**
char **drive*, **todrive*;

Description

The **dosassign** subroutine causes all references to the drive specified by the *drive* parameter to use the drive specified by the *todrive* parameter. The *drive* and *todrive* parameters are strings containing the names of drives as configured by **dosinit** or in a user profile. The names can be in either uppercase or lowercase and must not include the colon.

If the *drive* parameter is **NULL**, all assignments are reset to their initial state.

Once assigned, using the drive specified by the *drive* parameter is equivalent to using the drive specified by the *todrive* parameter. However, the **dospwd** subroutine does not perform this translation. It returns a path name that includes either the drive name passed to it as a parameter or the drive name passed to the **doschdir** subroutine.

The **dosassign** subroutine does not change the current drive.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosassign** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "doschdir" on page 3-72, and "dospwd" on page 3-96.

doschdir

Purpose

Changes the current DOS Services directory or current drive.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int doschdir (*path***)**
char ****path*;**

Description

The **doschdir** subroutine changes the current directory on the current drive to the directory specified by the *path* parameter, or changes the current drive to the drive specified in the *path* parameter. If the *path* parameter contains only a drive name, then only the current drive is changed. If the *path* parameter contains only a directory path name, then only the current directory on the current drive is changed. If the *path* parameter contains both a drive name and a directory path name, then the current drive and the current directory are both changed.

When the current drive is set to a drive that contains a DOS file system, the AIX current directory cannot follow along. Therefore, the current AIX directory is set to the special directory **/usr/dos/nulldir**, if it exists.

Normally, the user does not have write access to the **/usr/dos/nulldir** directory. Therefore, if a program aborts, the core dumps are suppressed. If you do not want this to happen, you must remove the **/usr/dos/nulldir** directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **doschdir** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "dospwd" on page 3-96, and "dosassign" on page 3-70.

doschmod

Purpose

Changes the mode of a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int doschmod (*path*, *mode***)**
char ****path*;**
long *mode***;**

Description

The **doschmod** subroutine changes the mode of the file specified by the *path* parameter to the mode specified by the *mode* parameter. (For information about modes, see "doscreate" on page 3-76.)

Return Value

Upon successful completion, a value of 0 is returned. If the **doschmod** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "doscreate" on page 3-76, and "dosopen" on page 3-94.

dosclose

Purpose

Closes a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosclose (*dosfile***)**
DOSFILE *dosfile***;**

Description

The **doscclose** subroutine closes the file descriptor specified by the *dosfile* parameter.

The *dosfile* parameter is a file descriptor obtained from a **dosopen**, **doscreate**, or **dosdup** subroutine.

Warning: DOS files are not implicitly closed when a process terminates. You must explicitly close all DOS files or you may lose data.

Return Value

Upon successful completion, a value of 0 is returned. If the **doscclose** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "doscreate" on page 3-76, "dosdup" on page 3-78, and "dosopen" on page 3-94.

doscreate

Purpose

Creates a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

DOSFILE doscreate (*path*, *mode***)**
char **path*;
long *mode*;

Description

The **doscreate** subroutine creates a DOS file with the path and name specified by the *path* parameter. The newly created file has the flags set as specified by the *mode* parameter. If the file specified by the *path* parameter already exists, the file is truncated to zero length and the mode and owner are unchanged.

The *mode* parameter is a 32-bit word containing flags. The low-order 12 bits are access permission flags. (For information about access permission flags, see "chmod" on page 2-18.)

If the file is contained in a DOS file system, the write-by-owner bit of the *mode* parameter is the only significant access flag. If it is 0, the read-only flag is set in the DOS directory of the file created.

Mode flags for emulating functions unique to DOS are defined in the **dos.h** header file. They may be logically OR'ed together from the following list:

M_HIDDEN If this flag of the *mode* parameter is set, the file is created as a *hidden* file. If the file is created in a DOS file system, the appropriate bit is set in the directory. If the file is created in an AIX file system, the file name is prefixed by a . (period). If a hidden file is created with the same file name as an existing normal file, the normal file is renamed.

M_SYSTEM If this flag of the *mode* parameter is set, and the file is created in a DOS file system, the **SYSTEM** attribute of the file is set. If the file is created in an AIX file system, this flag is ignored.

If the file is created in a DOS file system, the name of the file is translated to uppercase. If the file is created in an AIX file system, no translation takes place.

Warning: DOS files are not implicitly closed when a process terminates. You must explicitly close all DOS files or you may lose data.

Return Value

Upon successful completion, a non-**NULL** handle is returned. This handle is used in subsequent operations. The file is open for writing even if the mode does not permit writing. If the **doscreate** subroutine fails, a -1 is returned and **doserrno** is set to indicate the error.

The **doscreate** subroutine fails if one or more of the following are true:

- The user does not have write access to the directory containing the file.
- The user does not have write access to the file if the file already exists.
- The physical medium cannot be written to.
- No such device or address.
- No such device.

Related Information

In this book: "DOS services library" on page 3-65, "chmod" on page 2-18, and "dosopen" on page 3-94.

dosdup

Purpose

Duplicates a DOS Services file handle.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosdup (*dosfile***)**
DOSFILE *dosfile*;

Description

The **dosdup** subroutine returns a new file descriptor that indicates the same file and has the same open flags as the original file descriptor. The *dosfile* parameter is a file descriptor returned by **dosread**, **doscreate**, or **dosdup**. The file position is initially set to the same value as the original, but changes independently. The file descriptor returned is the lowest one available.

Return Value

Upon successful completion, a DOS Services file handle is returned. If the **dosdup** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "dosclose" on page 3-75. "doscreate" on page 3-76, and "dosopen" on page 3-94.

dosexecve

Purpose

Executes a program with a DOS path name.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosexecve (*path*, *argv*, *envp***)**
**char ****path*, ******argv*[ ], ******envp*[ ];

Description

The **dosexecve** subroutine invokes an AIX-executable program identified by a DOS path name. The **dosexecve** subroutine corresponds to the **execve** system call except that the *path* parameter is interpreted with respect to the configured DOS file system.

The *path* parameter must identify an AIX-executable file. It cannot refer to a DOS-executable file, such as an **.EXE**, **.COM**, or **.BAT** file. See the instructions for running a **.BAT** file on page 3-66.

The new program started by the **dosexecve** subroutine inherits the AIX run-time environment, which includes the AIX open file descriptors and other information. (See "fork" on page 2-46 and "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34 for a complete description of the AIX environment that is inherited after these system calls.)

However, the new program does *not* automatically inherit the DOS Services environment. See "dosunopen, dosreopen" on page 3-112 for details about passing the DOS Services environment to a child process.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosexecve** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fork" on page 2-46, "wait" on page 2-182, "dosinit" on page 3-85, and "dosunopen, dosreopen" on page 3-112.

dosfirst, dosnext

Purpose

Finds DOS files that match a pattern.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

char *dosfirst (*srch*, *pattern*, *mode***)**
**DOSFIND ***srch*;
**char ***pattern*;
long *mode*;

char *dosnext (*srch***)**
**DOSFIND ***srch*;

Description

The **dosfirst** and **dosnext** subroutines return a pointer to a memory area containing a file name that matches the pattern specified by the *pattern* parameter and that has the attributes specified by the *mode* parameter.

The *pattern* parameter is a file path name that can contain the pattern-matching characters ? (question mark) and * (asterisk).

The *srch* parameter points to a **DOSFIND** structure. That same **DOSFIND** structure should be passed to the **dosnext** subroutine on subsequent uses of the **dosnext** subroutine.

The *mode* parameter contains flags that specify files to include in the search. If the *mode* parameter is 0, directories, hidden files, and DOS system files are omitted from the search. You can use the following flags OR'ed together in any combination:

S_DIR	Includes directories in the search.
S_HIDDEN	Includes DOS/AIX hidden files in the search.
S_SYSTEM	Includes DOS system files in the search.
S_REG	Includes regular DOS files (all files other than directories, hidden files, or system files).
S_ALL	Includes all files (directories, hidden files, system files, and regular files).

DOSFIND is defined in the **dos.h** header file and has the following format:

```
typedef  long   DOSMODE;
typedef  short  DOSFILE;

typedef struct
{
   long     seek;
   int      count;
   long     *disk;
   int      mode;
   int      tnxtcl;
} dossrch;

typedef struct
{
   long     mode;
   char     path[128];
   char     *base;
   char     *extn;
   char     is_dos;
   DOSFILE  handle;
   short    index;
   dossrch  dos_srch;
} DOSFIND;
```

A return of **NULL** from either subroutine indicates that no more files matching the pattern can be found. If the search is terminated before the **NULL** return occurs, you should use the **free** subroutine to free the memory area returned from the last call.

Related Information

"DOS services library" on page 3-65.

dosfsync

Purpose

Synchronizes a specified DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosfsync (*dosfile***)**
DOSFILE *dosfile***;**

Description

The **dosfsync** subroutine guarantees that any changes to the file specified by the *dosfile* parameter has been written to the device on which the file exists when the subroutine returns. The use of the **dosfsync** subroutine has no detectable effect in a single process that runs to completion. It is useful in multi-processing applications and as a form of backup.

The *dosfile* parameter is an open file descriptor that was obtained from a **dosopen**, **doscreate**, or **dosdup** subroutine.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosfsync** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "fsync" on page 2-48, "DOS services library" on page 3-65, "doscreate" on page 3-76, "dosdup" on page 3-78, "dosopen" on page 3-94, and "doswrite" on page 3-116.

dosinit

Purpose

Initializes the DOS Services environment.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosinit ( )

Description

The **dosinit** subroutine initializes the run-time environment required by the DOS Services subroutines. Programs must call the **dosinit** subroutine before using any of the DOS Services subroutines.

The **dosinit** subroutine looks for the initialization information in the following places:

1. It attempts to read the information from the file named by **DOSENVT**. **DOSENVT** is a variable in the AIX environment. The **dos** command (or any parent process) creates this file by calling the **dosunopen** subroutine before starting a child process.
2. If the **DOSENVT** variable is not set, then **dosinit** configures DOS Services based on the AIX environment variables in the following list.
3. For the AIX variables that are not defined, the default values shown in the following are used.

AIX Variable	Default Value
DOS_NUL	/dev/null
DOS_CON	/dev/tty
DOS_COM1	/dev/tty0
DOS_COM2	/dev/tty1
DOS_AUX	/dev/tty0
DOS_LP0	/dev/lp0

DOS_LP1	/dev/lp1
DOS_LP2	/dev/lp2
.	.
.	.
.	.
DOS_LP7	/dev/lp7
DOS_A	/dev/fd0
DOS_B	/dev/fd0
DOS_C	$HOME
DOS_D	/
DOS_E	No default
DOS_F	No default
.	.
.	.
.	.
DOS_Z	No default.

Note: If **$HOME** is set to /, or if it is undefined, then the default directory for **DOS_C** is **/usr/dos**.

The value assigned to each of these variables must be either:

- The full AIX path name of an accessible device, file, or directory, or
- A string in the form |*command*, where *command* is an AIX command. The specified command receives, as its standard input, the data that DOS Services programs write to the corresponding logical device.

The **dosinit** subroutine also uses the following environment variable:

DOSDISK Specifies the initial default drive for DOS Services. If **DOSDISK** is not set, then **dosinit** searches sequentially from **A:** to **Z:** and sets the default drive to the first valid file system found.

The AIX environment variables that specify the DOS Services configuration can be set in the system **/etc/profile** file, in the user's **$HOME/.profile**, or directly from the AIX command line. See the **sh** command in *AIX Operating System Commands Reference* for more information about setting AIX environment variables, which are also called ***shell variables***.

The AIX environment variables are accessed only once during a single session started by the **dos** command. From then on, internal configuration tables are used.

Return Value

Upon successful completion **dosinit** returns a value of 0. Otherwise a value of -1 is returned, and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "dosread" on page 3-98, "dosunopen, dosreopen" on page 3-112, and "doswrite" on page 3-116

The **dos** and **sh** commands in *AIX Operating System Commands Reference.*

doslock

Purpose

Locks or unlocks a region in a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int doslock (*dosfile*, *offset*, *length*, *code***)**
DOSFILE *dosfile*;
int *offset*, *length*, *code*;

Description

The **doslock** subroutine provides a simple mechanism for disallowing read/write access by other processes to the file whose handle is specified by the *dosfile* parameter. If *code* is **L_LOCK**, and no part of the region is already locked by another process, then the region is locked. If *code* is **L_UNLOCK**, and the exact region has already been locked by the current process, then the region is unlocked.

Return Value

If the **doslock** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error. For **L_LOCK**, the error means that the lock failed, either because the region was already locked, or because the lock list is full. For **L_UNLOCK**, the error means that the region was not locked.

Related Information

In this book: "DOS services library" on page 3-65, "dosread" on page 3-98, and "doswrite" on page 3-116.

dosmkdir

Purpose

Creates a directory.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosmkdir (*path***)**
**char ****path*;

Description

The **dosmkdir** subroutine creates a directory using the path specified by the *path* parameter. All components of the *path* parameter except the last component must already exist.

If creating an AIX directory, the **dosmkdir** subroutine forks and executes the AIX **/bin/mkdir** command, which creates the directory. This is done because only a process with an effective user ID of superuser can create an AIX directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosmkdir** subroutine fails, then it returns a nonzero value and sets **doserrno** to indicate the error.

Certain failures of the **mkdir** command also cause an error message to be written to the standard error output.

Related Information

In this book: "DOS services library" on page 3-65 and "dosrmdir" on page 3-102.

The **mkdir** command in *AIX Operating System Commands Reference.*

dosmktemp

Purpose

Creates a DOS temporary file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

DOSFILE dosmktemp ( )

Description

The **dosmktemp** subroutine creates and opens a temporary file. The file is open for reading and writing. The access permission flags are set so that the owner has read-write permission and all others have no permission.

The file is created in the **/tmp** directory with a name that includes the process ID of the running process and a serial number. The **/tmp** directory does not need to be accessible through the DOS file system configuration.

Return Value

Upon successful completion, a non-**NULL** file handle is returned. This handle is to be used in subsequent operations.

If the **dosmktemp** subroutine fails, -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “DOS services library” on page 3-65, “doscreate” on page 3-76, and “dosopen” on page 3-94.

dosopen

Purpose

Opens a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

```
#include <dos.h>

DOSFILE dosopen (path, oflag, mode)
char *path;
int oflag;
long mode;
```

Description

The **dosopen** subroutine opens the file specified by the *path* parameter. The *oflag* parameter specifies the type of open. The *mode* parameter specifies the access mode of the file if a new file is created.

The *oflag* parameter is constructed by logically OR-ing one or more of the following values:

DO_RDONLY Open for reading only.

DO_WRONLY Open for writing only.

DO_RDWR Open for reading and writing.

DO_APPEND If set, the file pointer is set to the end of the file prior to each write operation.

DO_CREAT If the file does not exist, create it. Use *mode* to establish the protection mode of the new file. (For information on creating a DOS file, see "doscreate" on page 3-76.)

DO_TRUNC If the file exists, truncate it. Otherwise begin writing at the end of file.

DO_EXCL	If the file already exists, the **dosopen** subroutine fails.
DO_ASCII	Interpret the file as an ASCII text file. (For information about ASCII files, see "dosread" on page 3-98 and "doswrite" on page 3-116.)

Note: Only one of **DO_RDONLY**, **DO_WRONLY**, and **DO_RDWR** can be specified. The others can be used in any combination.

If the file being opened is on a DOS file system, the name file name given is translated to uppercase. If the file system is an AIX file system, no translation takes place.

Warning: DOS files are not implicitly closed when a process terminates. You must explicitly close all DOS files or you may lose data.

Return Value

Upon successful completion, a DOS Services file handle is returned. This handle is used in subsequent operations. If the **dosopen** subroutine fails, -1 value is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "dosclose" on page 3-75, "doscreate" on page 3-76, "dosdup" on page 3-78, "dosread" on page 3-98, "dosseek" on page 3-104, and "doswrite" on page 3-116.

dospwd

Purpose

Gives the full path name of the current directory.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

char *dospwd (*drive***)**
char ***drive*;**

Description

The **dospwd** subroutine returns a pointer to the area of memory that contains the null-terminated name of the current directory for the drive specified by the *drive* parameter. If the *drive* parameter is a valid drive name (such as "A:" or "B:"), then the current directory for that drive is returned. If the *drive* parameter is **NULL**, then the current directory for the current drive is returned.

Return Value

Upon successful completion, a pointer to a string specifying the full path name of the current directory is returned. The memory for the name is allocated with the **malloc** subroutine and should be deallocated with the **free** subroutine. If the **dospwd** subroutine fails, a **NULL** pointer is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "dosassign" on page 3-70, and "doschdir" on page 3-72.

dosread

Purpose

Reads from a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosread (*dosfile*, *buf*, *n***)**
DOSFILE *dosfile*;
char **buf*;
int *n*;

Description

The **dosread** subroutine reads the number of bytes specified by the *n* parameter. The *dosfile* parameter specifies the handle of the file from which the bytes are to be read. The bytes read from the file are written into the buffer pointed to by the *buf* parameter.

Reading begins from the current position in the file. The current position of the file is incremented by the number of bytes read.

When attempting to read from a region that has been locked with the **doslock** subroutine, the **dosread** subroutine retries three times at one second intervals. If the region is still locked, an error occurs and **-1** is returned. The proper method for using locks is not to rely on being denied I/O access, but to attempt to lock the desired region and then examine the return code.

If the **DO_ASCII** flag was set when the file was opened, and if the file is located on a DOS file system, then the **dosread** subroutine translates the DOS ASCII data to AIX ASCII format. If **DO_ASCII** was not set when the file was opened, or if the file is located on an AIX file system, then this translation does not take place.

When DOS ASCII translation is being performed, the **dosread** subroutine removes the ASCII CR characters, thus changing the CR-LF sequence used by DOS to the `'\n'` (new-line character) that AIX uses. An end-of-file condition occurs when **Ctrl-Z** is encountered. The **dosread** subroutine does not return the **Ctrl-Z** character as part of the data.

Return Value

Upon successful completion, the number of bytes actually read (after DOS translation, if any) is returned. If an end-of-file is read, a value of 0 is returned. If the **dosread** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “DOS services library” on page 3-65, “dosclose” on page 3-75, “doscreate” on page 3-76, “doslock” on page 3-88, “dosopen” on page 3-94, “dosseek” on page 3-104, and “doswrite” on page 3-116.

dosrename

Purpose

Renames a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosrename (*oldpath*, *newfile***)**
**char ****oldpath*, ***newfile*;

Description

The **dosrename** subroutine changes the name of a DOS file. The file to be renamed is specified by the path name pointed to by the *oldpath* parameter. The new name for that file is specified by the *newfile* parameter. The *newfile* parameter must be a simple file name and optional extension.

The **dosrename** subroutine must not be used to move a file from one directory to another directory. The **dosrename** subroutine fails if the file name specified by the *newfile* parameter already exists in the directory specified by the *oldpath* parameter.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosrename** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: “DOS services library” on page 3-65, “doscreate” on page 3-76, “dosopen” on page 3-94, and “dosunlink” on page 3-110.

dosrmdir

Purpose

Removes a DOS Services directory.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosrmdir (*path***)**
char ***path*;**

Description

The **dosrmdir** subroutine removes the directory specified by the *path* parameter. A directory must be empty before it can be removed. To remove a directory, the current process must have write access permission to the directory and to its parent directory.

If an AIX directory is to be removed, the **dosrmdir** subroutine forks and executes the AIX **/bin/rmdir** command, which removes the directory. This is done because only a process with an effective user ID of superuser can remove an AIX directory.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosrmdir** subroutine fails, it returns a nonzero value and sets **doserrno** to indicate the error. Certain failures of the **rmdir** command also cause an error message to be written to the standard error output.

Related Information

In this book: "DOS services library" on page 3-65 and "dosmkdir" on page 3-90.

The **rmdir** command in *AIX Operating System Commands Reference.*

dosseek

Purpose

Moves the DOS file read/write pointer.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosseek (*dosfile*, *offset*, *whence***)**
DOSFILE *dosfile*;
long *offset*;
int *whence*;

Description

The **dosseek** subroutine moves the current position pointer in the file specified by the *dosfile* parameter.

- If the *whence* parameter is 0, the pointer is set to the position specified by the *offset* parameter.
- If the *whence* parameter is 1, the pointer is incremented by the number of bytes specified by the *offset* parameter.
- If the *whence* parameter is 2, the pointer is set to the size of the file plus the number of bytes specified by the *offset* parameter.

If the **dosseek** subroutine has been issued on a file that was opened with the **DO_ASCII** flag set, the *offset* parameter must be 0. Otherwise, the results are unpredictable.

Return Value

Upon successful completion, the file pointer value is returned. If the **dosseek** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Diagnostics

The **dosseek** subroutine fails and the file pointer remains unchanged if one or more of the following are true:

EBADF *dosfile* is not an open file descriptor.

ESPIPE *dosfile* is associated with a pipe or FIFO.

EINVAL *whence* is not 0, 1 or 2. This also causes a **SIGSYS** signal.

EINVAL The resulting file pointer would be negative.

Related Information

In this book: "DOS services library" on page 3-65, "doscreate" on page 3-76, "dosdup" on page 3-78, and "dosopen" on page 3-94.

dosstat, dosfstat

Purpose

Gets the status of a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosstat (*path*, *buf***)**
char **path*;
DOSSTAT **buf*;

int dosfstat (*dosfile*, *buf***)**
DOSFILE *dosfile*;
DOSSTAT **buf*;

Description

The **dosstat** subroutine gets the status of the file named by the *path* parameter and stores that status into the memory area pointed to by the *buf* parameter. Read, write, and execute permissions are not required of the named file, but all directories included in the path name must be searchable.

The **dosfstat** subroutine gets the status of an open file named by the *dosfile* parameter and stores that status into the memory area pointed to by the *buf* parameter.

The **DOSSTAT** structure is defined in the **dos.h** file and has the following form:

```
typedef struct
{
   char    st_drive_id, st_filetype;
   long    st_mode, st_ino, st_dev, st_rdev,
           st_nlink, st_uid, st_gid, st_size;
   time_t  st_atime, st_mtime, st_ctime;
} DOSSTAT;
```

The **st_drive_id** field is set to the logical drive on which the file exists. The **st_filetype** field is set to the following values:

u For an AIX file.
d For a DOS file.
t For a tty file.
o For any other type of file.

For non-DOS files, all other fields are identical to the fields defined in "stat.h" on page 5-69. For DOS files, the fields have the following meanings:

st_mode If the M_DIRECTORY bit is set, the file is a directory. All other bits correspond to the *mode* bits as defined in the **doscreate** subroutine.

st_ino The number of the first cluster in the file.

st_rdev Contains the attribute byte from the DOS directory entry.

st_size The size in bytes of the file as set in the DOS directory entry.

st_mtime The time the file was last modified, in AIX format. (For information about this format, see "time" on page 2-164 and "ctime, localtime, gmtime, asctime, tzset" on page 3-46.)

st_dev Meaningless for DOS files.

st_nlink Meaningless for DOS files.

st_uid Meaningless for DOS files.

st_gid Meaningless for DOS files.

st_atime Meaningless for DOS files.

st_ctime Meaningless for DOS files.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosstat** or **dosfstat** subroutines fail, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "stat, fstat" on page 2-159, and "dosustat" on page 3-114.

dostouch

Purpose

Changes the modification date of a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dostouch (*path*, *date***)**
char **path*;
long *date*;

Description

The **dostouch** subroutine changes the time of the last modification of the file specified by the *path* parameter to the time specified by the *date* parameter. The *time* parameter contains a time in a format like that returned by the **time** system call (see "time" on page 2-164).

If the *path* parameter identifies a file on an AIX file system, the program **/bin/touch** is invoked to change the time.

Return Value

Upon successful completion, a value of 0 is returned. If the **dostouch** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "time" on page 2-164, "ctime, localtime, gmtime, asctime, tzset" on page 3-46, "DOS services library" on page 3-65, and "dosstat, dosfstat" on page 3-106.

dosunlink

Purpose

Unlinks (deletes) a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosunlink (*path***)**
char ****path*;**

Description

The **dosunlink** subroutine deletes the directory entry named by the *path* parameter.

The file is removed from the file system if it is a DOS file or if it is an AIX file with no other links to it.

The **dosunlink** subroutine fails if the invoking process does not have write access to the file being removed and to the directory in which it is contained. It is successful if the file is in use, but the file is not deleted until all users have closed it.

The directory entry is removed upon return to **dosunlink**, but all users who had the file open when **dosunlink** was called still have a local copy for their use.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosunlink** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "doscreate" on page 3-76, "dosopen" on page 3-94, and "dosrename" on page 3-100.

dosunopen, dosreopen

Purpose

Passes the DOS Services environment from a parent process to a child process.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosunopen ( )	**int dosreopen ( )**

Description

The **dosunopen** and **dosreopen** subroutines provide a means to pass the current DOS Services environment to a child process. The following sequence accomplishes this task:

1. The parent process calls the **dosunopen** subroutine to construct a file that describes the current state of the DOS Services environment. **dosunopen** also sets the **DOSENVT** variable (in the AIX environment) to the name of this file.
2. The parent process then issues a **fork** system call to start a child process. The child process automatically inherits copies of its parent's AIX open file descriptors, but not the DOS file descriptors.
3. The child process calls the **dosexecve** subroutine (or one of the **exec** system calls) to run a new program. This program calls the **dosinit** subroutine, which initializes the child's DOS Services environment from the file named by the **DOSENVT** variable. The child process now has access to the DOS files opened by its parent.
4. The parent process invokes the **wait** system call to wait for the child to finish running its program.
5. When the child has finished, the parent process calls the **dosreopen** subroutine to reopen the original DOS Services environment and to delete the environment file created by **dosunopen**.

If a program attempts to reopen a file on a removable diskette and that diskette is not inserted into the machine, then the **dosreopen** subroutine prompts you to reinsert the correct diskette.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosunopen** or **dosreopen** subroutine fails, then it returns a value of -1 and sets **doserrno** to indicate the error.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "fork" on page 2-46, "wait" on page 2-182, "DOS services library" on page 3-65, "dosinit" on page 3-85, "dosexecve" on page 3-79, "dosopen" on page 3-94, and "getenv, NLgetenv" on page 3-208.

dosustat

Purpose

Gets the status of a given DOS Services device.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int dosustat (*device*, *buf***)**
char **device*;
DOSUSTAT **buf*;

Description

The **dosustat** subroutine gets the status of the DOS Services device specified by the *device* parameter and returns that information to the area of memory specified by the *buf* parameter. The *device* parameter is a pointer to a string that contains the name of a DOS Services device.

The **DOSUSTAT** structure is defined in **/usr/include/dos.h** and has the following format:

```
typedef struct
{
   char  upath[128];   /* Device or file specified in configuration */
   char  umount[128];  /* AIX directory mounted on it, or NULL */
   char  volume[32];   /* Volume name */
   int   freespace;    /* Number of free bytes */
   char  fstype;       /* AIX file system='u', DOS file system='d' */
} DOSUSTAT;
```

The **umount** structure member is not **NULL** only when the *upath* field identifies a device name (such as **/dev/fd0**), and that device contains a mountable AIX file system.

The **volume**, **freespace**, and **fstype** members are set only when the *device* parameter is a disk name (a single letter followed by a colon). If the *device* is a removable disk, **umount** may read it to determine if it is a valid DOS or AIX file system.

Return Value

Upon successful completion, a value of 0 is returned. If the **dosustat** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "ustat" on page 2-178, "DOS services library" on page 3-65, and "dosstat, dosfstat" on page 3-106.

doswrite

Purpose

Writes to a DOS file.

Library

DOS Services Library (**libdos.a**)

Syntax

#include <dos.h>

int doswrite (*dosfile*, *buf*, *n***)**
DOSFILE *dosfile*;
char **buf*;
int *n*;

Description

The **doswrite** subroutine writes the number of bytes specified by the *n* parameter to the file whose handle is specified by the *dosfile* parameter. The bytes are written from the buffer pointed to by the *buf* parameter.

Writing begins from the current position in the file. The current position is incremented by the number of bytes written. When attempting to write to a region that has been locked with the **doslock** subroutine, the doswrite subroutine retries three times at one second intervals. If the region is still locked, an error occurs and **-1** is returned. The proper method for using locks is not to rely on being denied I/O access, but to attempt to lock the desired region and then examine the return code.

If the file was opened with the **DO_APPEND** flag set, then the file pointer is set to the end of the file prior to each write.

If the **DO_ASCII** flag was set when the file was opened, and if the file is located on a DOS file system, then the **doswrite** subroutine translates the AIX ASCII data to DOS ASCII format. If **DO_ASCII** was not set when the file was opened, or if the file is located on an AIX file system, then this translation does not take place.

When DOS ASCII translation is being performed, the **doswrite** subroutine inserts an ASCII CR character before each LF character, thus changing each AIX `'\n'` (new-line character) to the CR-LF sequence that DOS uses to indicate a new line. A **Ctrl-Z** character is written as the last character when the file is closed.

If a write requests that more translated bytes be written than there is room for in the file, then only as many bytes as there is room for are written. The next request to write more than zero bytes returns an error.

Return Value

Upon successful completion, a number equal to the number of bytes written before translation is returned. If the **doswrite** subroutine fails, a value of -1 is returned and **doserrno** is set to indicate the error.

Related Information

In this book: "DOS services library" on page 3-65, "dosclose" on page 3-75, "doscreate" on page 3-76, "doslock" on page 3-88, "dosopen" on page 3-94, "dosread" on page 3-98, and "dosseek" on page 3-104.

drand48

Purpose

Generates uniformly distributed pseudo-random number sequences.

Library

Standard C Library (**libc.a**)

Syntax

double drand48 ( )

double erand48 (*xsubi*)
unsigned short *xsubi*[3];

long lrand48 ( )

long nrand48 (*xsubi*)
unsigned short *xsubi*[3];

long mrand48 ( );

long jrand48 (*xsubi*)
unsigned short *xsubi*[3];

void srand48 (*seedval*)
long *seedval*;

unsigned short *seed48 (*seed16v*)
unsigned short *seed16v*[3];

void lcong48 (*param*)
unsigned short *param*[7];

Description

This family of subroutines generates pseudo-random numbers using the linear congruential algorithm and 48-bit integer arithmetic.

The **drand48** and **erand48** subroutines return nonnegative double-precision floating-point values uniformly distributed over the range of *y* values such that $0.0 \leq y < 1.0$.

The **lrand48** and **nrand48** subroutines return nonnegative long integers uniformly distributed over the range of *y* values such that $0 \leq y < 2^{31}$.

The **mrand48** and **jrand48** subroutines return signed long integers uniformly distributed over the range of *y* values such that $-2^{31} \leq y < 2^{31}$.

The **srand48**, **seed48** and **lcong48** subroutines initialize, the random-number generator. Programs should invoke one of them before calling **drand48**, **lrand48** or **mrand48**. (Although it is not recommended practice, constant default initializer values are supplied automatically if the **drand48**, **lrand48** or **mrand48** subroutines are called without first calling an initialization subroutine.) The **erand48**, **nrand48** and **jrand48** subroutines do not require that an initialization subroutine to be called first.

All the subroutines work by generating a sequence of 48-bit integer values, X_i, according to the linear congruential formula:

$$X_{n+1} = (aX_n + c)_{\mathrm{mod}\, m} \quad n \geq 0$$

The parameter $m = 2^{48}$; hence 48-bit integer arithmetic is performed. Unless the **lcong48** subroutine has been called, the multiplier value a and the addend value c are:

$$a = 5\mathrm{DEECE66D}_{16} = 273673163155_8$$

$$c = \mathrm{B}_{16} = 13_8$$

The value returned by the **drand48**, **erand48**, **lrand48**, **nrand48**, **mrand48**, and **jrand48** subroutines is computed by first generating the next 48-bit X_i in the sequence. Then the appropriate number of bits, according to the type of data item to be returned, are copied from the high-order (most significant) bits of X_i and transformed into the returned value.

The **drand48**, **lrand48** and **mrand48** subroutines store the last 48-bit X_i generated into an internal buffer; that is why they must be initialized prior to being invoked.

The **erand48**, **nrand48** and **jrand48** subroutines require the calling program to provide storage for the successive X_i values in the array pointed to by the *xsubi* parameter. That is why these routines do not have to be initialized; the calling program merely has to place the desired initial value of X_i into the array and pass it as a parameter.

By using different parameters, the **erand48**, **nrand48**, and **jrand48** subroutines allow separate modules of a large program to generate several *independent* sequences of pseudo-random numbers. In other words, the sequence of numbers that one module generates does *not* depend upon how many times the subroutines are called by other modules.

The initializer subroutine **srand48** sets the high-order 32 bits of X_i to the 32 bits contained in its parameter. The low order 16 bits of X_i are set to the arbitrary value $330\mathrm{E}_{16}$.

The initializer subroutine **seed48** sets the value of X_i to the 48-bit value specified in the array pointed to by the *seed16v* parameter. In addition, **seed48** returns a pointer to a 48-bit internal buffer that contains the previous value of X_i. that is used only by **seed48**. The returned pointer allows you to restart the pseudo-random sequence at a given point. Use the pointer to copy the previous X_i value into a temporary array. Later you can call **seed48** with a pointer to this array to resume where the original sequence left off.

The **lcong48** subroutine specifies the initial X_i value, the multiplier value a, and the addend value c. The parameter array elements *param*[0-2] specify X_i, *param*[3-5] specify the multiplier a, and *param*[6] specifies the 16-bit addend c. After **lcong48** has been called, a subsequent call to either **srand48** or **seed48** restores the standard a and c as specified previously.

Related Information

In this book: "rand, srand" on page 3-317.

drsname, drsnidd

Purpose

Returns the associated node ID for a given node nickname or the associated nickname for a given node ID.

Library

IPC Library (**libipc.a**)

Syntax

#include <drs.h>

int drsname (*nickname*, *nid*)

char **nickname*;
long **nid*;

int drsnidd (*nid*, *nickname*)

long *nid*;
char **nickname*;

Description

The **drsname** subroutine returns the associated node ID for a given nickname. The *nickname* parameter points to a null-terminated string containing the nickname for which the node ID is to be returned. The *nid* parameter points to the location of the returned node ID.

The **drsnidd** subroutine returns the associated node nickname for a given node ID. The *nid* parameter contains the node ID for which the nickname is to be returned. The *nickname* parameter is a null-terminated string that points to the returned nickname. The calling process must reserve enough memory to contain the returned information, which is a maximum of 14 characters plus the trailing **NULL**.

Return Value

Upon successful completion, these functions return a value of 0. If an error occurs, they return one of the following negative values:

DRS_ACCES The required access permissions were denied.

DRS_BADLEN An incorrect parameter was supplied.

DRS_NOREC	No record was found.
DRS_IO	An input/output error occurred.
DRS_AGAIN	Unable to start Profile Services.
DRS_BADF	An incorrect file descriptor was supplied.
DRS_BADK	An incorrect index key was supplied.
DRS_BDMSF	An incorrect file or table was supplied.
DRS_BOF	The beginning of the file was encountered.
DRS_DEADLK	A deadlock was detected.
DRS_EOF	The end of the file was encountered.
DRS_FAULT	An incorrect address was supplied.
DRS_FBIG	The maximum file size was exceeded.
DRS_IDRM	Identifier removed.
DRS_INTENT	Intentions denied.
DRS_ISDIR	A write to a directory was attempted.
DRS_MFILE	Too many files, tables, or indexes were open.
DRS_NFILE	The file table overflowed.
DRS_NOENT	No file or directory was found.
DRS_NOMEM	No memory is available.
DRS_NOSPC	No space is available on the device.
DRS_NOTDIR	Not a directory.
DRS_NOTIDX	Not an index.
DRS_PANIC	Abnormal termination occurred.
DRS_RCVRY	File needs recovery.
DRS_RECLEN	Record length is invalid.
DRS_ROFS	The file system to be accessed is read-only.

Related Information

Managing the AIX Operating System.

ecvt, fcvt, gcvt

Purpose

Converts a floating-point number to a string.

Library

Standard C Library (**libc.a**)

Syntax

```
char *ecvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt (value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt (value, ndigit, buf)
double value;
int ndigit;
char *buf;
```

Description

The **ecvt**, **fcvt**, and **gcvt** subroutines convert floating-point numbers to strings.

The **ecvt** subroutine converts the *value* parameter to a null-terminated string and returns a pointer to it. The *ndigit* parameter specifies the number of digits in the string. The low-order digit is rounded. **ecvt** sets the **int** pointed to by the *decpt* parameter to the position of the decimal point relative to the beginning of the string. (A negative number means the decimal point is to the left of the digits given in the string). The decimal point itself is not included in the string. The **ecvt** subroutine also sets the **int** pointed to by the *sign* parameter to a non-zero value if the *value* parameter is negative, and sets it to 0 otherwise.

The **fcvt** subroutine functions identically to **ecvt**, except that it rounds the correct digit for outputting *ndigit* digits in FORTRAN F-format.

The **gcvt** subroutine converts the *value* parameter to a null-terminated string, stores it in the array pointed to by the *buf* parameter, and then returns *buf*. **gcvt** attempts to produce a string of *ndigit* significant digits in FORTRAN F-format. If this is not possible, then E-format is used. **gcvt** suppresses trailing zeroes. The string is ready for printing, complete with minus sign, decimal point, or exponent, as appropriate.

The **ecvt**, **fcvt**, and **gcvt** subroutines represent the following special values that are specified in ANSI/IEEE standard 754-1985 for binary floating-point arithmetic:

Quiet NaN	QNaN
Signalling NaN	SNaN
±∞	INF

The sign associated with each of these values is stored into the *sign* parameter. Note, also, that 0 can be positive or negative.

Warning: All three subroutines store the strings in a static area of memory whose contents are overwritten each time one of the subroutines is called.

Related Information

In this book: "a64l, l64a" on page 3-4, "frexp, ldexp, modf" on page 3-194, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300, and "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325.

end, etext, edata

Purpose

Defines the last location of a program.

Library

None

Syntax

```
extern end;
extern etext;
extern edata;
```

Description

The external names **end**, **etext**, and **edata** are defined by the loader for all programs. They are not subroutines, but identifiers associated with the following addresses:

etext The first address following the program text
edata The first address following the initialized data region
end The first address following the data region that is not initialized.

The ***program break*** is the first location beyond the data. When a program begins running, this location coincides with **end**. However, many factors can change the program break, including:

- The **brk** system call
- The **malloc** subroutine
- The standard input/output subroutines
- The **-p** flag on the **cc** command.

Therefore, use **sbrk(0)**, not **end**, to determine the program break.

Related Information

In this book: "brk, sbrk" on page 2-14, "malloc, free, realloc, calloc" on page 3-236, and "standard i/o library" on page 3-342.

The **cc** command in *AIX Operating System Commands Reference.*

erf, erfc

Purpose

Computes the error and complementary error functions.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double erf (*x*)
double *x*;

double erfc (*x*)
double *x*;

Description

The **erf** subroutine returns the error function of *x*, defined as:

$$\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$$

The **erfc** subroutine returns 1.0 - **erf**(*x*). The **erfc** subroutine is provided because of the extreme loss of relative accuracy if **erf**(*x*) is called for large values of *x* and the result is subtracted from 1.0. For example, 12 decimal places are lost when calculating (1.0 - **erf**(5)).

Related Information

In this book: "exp, log, log10, pow, sqrt" on page 3-128.

errunix

Purpose

Logs application errors.

Library

Run-time Services Library (**librts.a**)

Syntax

int errunix (*buf*, *cnt***)**
char **buf*;
unsigned int *cnt*;

Description

The **errunix** subroutine invokes the application error device driver to record an error log entry. **errunix** is a C run-time subroutine. Device drivers should use the **errsave** subroutine to log error messages.

If the error device driver is not open, **errunix** opens it. Then the error log entry is written to it.

The *buf* parameter points to a buffer that contains the following information:

1. A word (**int**) that contains the **class**, **subclass**, **mask**, and **type** of the message, as defined in the discussion of "error" on page 6-15
2. An **int** that specifies the number of words of dependent data for the error log entry, including this **int** itself
3. Words that contain the dependent information for the error log entry. The number of dependent data words must be one less than the word count specified immediately before them.

The other fields of the error log header (length, date and time, time extended, node name, and virtual machine ID) are supplied for you automatically.

The *cnt* parameter specifies the number of bytes in the buffer pointed to by *buf*. *cnt* must be a multiple of 4.

Return Value

Upon successful completion, a value of 0 is returned. If the **errunix** subroutine fails, an error message is written to the standard error output, and a value of -1 is returned.

File

/dev/error

Related Information

In this book: "error" on page 6-15, and "errsave" on page C-31.

exp, log, log10, pow, sqrt

Purpose

Computes exponential, logarithm, power, and square root functions.

Library

Math Library (**libm.a**)

Syntax

```
#include <math.h>

double exp (x)
double x;

double log (x)
double x;

double log10 (x)
double x;

double pow (x, y)
double x, y;

double sqrt (x)
double x;
```

Description

The **exp** subroutine to returns e^x.

The **log** subroutine returns the natural logarithm of *x*, ln *x*. The value of *x* must be positive.

The **log10** subroutine returns the logarithm base 10 of *x*, $\log_{10} x$. The value of *x* must be positive.

The **pow** subroutine returns x^y. The values of *x* and *y* may not both be 0. If *x* is negative or 0, then *y* must be an integer.

The **sqrt** subroutine returns the square root of *x*. The value of *x* can not be negative.

Diagnostics

The **exp**, **log**, **log10**, and **sqrt** subroutines can perform either of the following types of error handling. The **pow** subroutine always handles errors according to the first method. Both types of error handling allow you to define special actions to be taken when an error occurs.

1. By default, **matherr** error handling is performed, as described on page 3-238. The default error-handling procedures for these subroutines are as follows:

   exp If the correct value overflows, **exp** returns **HUGE** and sets **errno** to **ERANGE**.

   log If *x* is negative or 0, then **log** returns the value **-HUGE**, sets **errno** to **EDOM**, and writes an error message to the standard error output.

   log10 If *x* is negative or 0, then **log10** returns the value **-HUGE**, sets **errno** to **EDOM**, and writes an error message to the standard error output.

   pow If *x* is negative or 0 and *y* is not an integer, or if *x* and *y* are both 0, then **pow** returns the value 0, sets **errno** to **EDOM**, and writes an error message to the standard error output. If the correct value overflows, **pow** returns **HUGE** and sets **errno** to **ERANGE**.

   sqrt If *x* is negative, then **sqrt** returns the value 0, sets **errno** to **EDOM**, and writes an error message to the standard error output.

2. For **exp**, **log**, **log10**, and **sqrt**, exception handling can also be performed according to ANSI/IEEE standard 754-1985 for binary floating-point arithmetic, as discussed under "Exception Handling" on page 3-186. To select ANSI/IEEE exception handling, define the **_C_func** preprocessor variable. You can do this by inserting the statement **#define _C_func** before the **#include <math.h>**, or by specifying the **-D_C_func** flag to the **cc** command when compiling the program.

   If a hardware floating-point processor is installed in your system, then using this option can provide greater performance in addition to IEEE exception handling. Defining **_C_func** causes the **math.h** header file to define macros that make the names **exp**, **log**, **log10**, and **sqrt** appear to the compiler as **_C_exp**, **_C_log**, **_C_log10**, and **_C_sqrt**, respectively. These special names instruct the C compiler to generate code that avoids the overhead of the math library subroutines and issues compatible-mode floating-point calls directly. See "fpfp" on page 3-170 for information about compatible mode.

Related Information

In this book: "fpfp" on page 3-170, "Exception Handling" on page 3-186, "hypot" on page 3-229, "matherr" on page 3-238, and "sinh, cosh, tanh" on page 3-337.

extended curses library

Purpose

Controls cursor movement and windowing.

Library

Extended Curses Library (**libcur.a**)

Syntax

#include <cur01.h>

Description

The Extended Curses subroutines control input and output to a work station, performing optimized cursor movement, windowing, and other functions. This package is based on the **curses** subroutine package, which is included in most UNIX-compatible systems. The **curses** subroutines are also included in AIX for complete compatibility with existing programs (see "curses" on page 3-51).

The enhancements provided by Extended Curses include:

- A wider range of display attributes
- Generalized drawing of boxes
- Terminal-independent input data processing
- Extended window control
- Pane, panel, and field concepts
- Support for extended characters
- Handling of locator input.

Terminology

window The internal representation of what a portion of the display may look like at some point in time. Windows can be any size from the entire display screen to a single character.

screen A window that is large as the display screen. A screen named **stdscr** is automatically provided.

terminal Sometimes called a ***terminal screen***. A special screen that is the Extended Curses package's understanding of what the work station's display screen currently looks like. The terminal screen is identified by a window named **curscr**, which should not be accessed directly by the user. Instead, changes should be made to **stdscr** (or a user-defined screen) and then **refresh** (or **wrefresh**) should be called to update the terminal.

presentation space
The array that contains the data and attributes associated with a window.

pane An area of the display that shows all or part of the data contained in a presentation space associated with that pane.

active pane
The pane in which the text cursor is positioned. A pane must be active before you can do input.

panel A group of one or more panes that are treated as a unit. The panes of a panel are displayed together, erased together, and usually represent a unit of information to a person using the application. A panel is represented on the display as a rectangular area that is tiled (completely filled) with panes.

field An area in a presentation space into which the program accepts input.

extended character
A character other than 7-bit ASCII that can be represented in either 1 or 2 bytes. (See "data stream" on page 5-5.)

NLSCHAR
A data type that represents a character from code page P0, P1, or P2. It is defined to be equivalent to **unsigned short**. A single **NLSCHAR** variable can contain either a 1-byte or a 2-byte character. The 1-byte characters are stored in the low-order byte with the high-order byte set to 0; this is the same way that they are stored as integers. The 2-byte characters are stored with the single-shift control code in the high-order byte and the character data code in the low-order byte. This data type has no relation to the **NLchar** data type.

See the discussion of Extended Curses in *AIX Operating System Programming Tools and Interfaces*, and "Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System* for more detailed information about these concepts.

Linking the Extended Curses Routines

The Extended Curses routines also call **terminfo** subroutines, which are located in the Curses Library (**libcurses.a**). Therefore, compile programs that use Extended Curses routines with the flags **-lcur -lcurses**.

Header Files

- The **cur00.h** header file replaces **curses.h** when converting programs that use the original **curses** package to Extended Curses.
- All of the routines require the **cur01.h** header file.
- The key codes returned by **getch** are defined in **cur02.h**.
- The **cur03.h** header file defines attribute priority codes, and is not needed by application programs.
- The **unctrl** routine requires **cur04.h**.
- The routines that manage panes and panels (the routines whose names begin with **ec**) also require the **cur05.h** header file.

Naming Conventions

The new routines added to the original **curses** package begin with the letters **ec**.

Many routines operate on **stdscr**, the standard screen, by default. Corresponding routines that allow you to specify a window have the same name, prefixed with the letter **w**. For example, **addch** adds a character to **stdscr**, while **waddch** allows you to specify the window. Sometimes a routine beginning with **p** also exists, such as **paddch**, which allows you to specify a pane.

Some routines also allow you to specify cursor movement with the action to be performed. These routines have a prefix of **mv**. Thus, **addch** becomes **mvaddch**, **waddch** becomes **mvwaddch**, and **paddch** becomes **mvpaddch**. Each of these routines is equivalent to calling **move** or **wmove** before performing the operation.

The various prefixed forms of the routines are implemented as macros. In each case, the routine beginning with **w** is the base subroutine from which the others are defined.

Parameters

The following declarations serve for all of the routines:

> **char** *ch *string*;
> **NLSCHAR** *xc*;
> **int** *line*, *col*, *firstline*, *firstcol*;
> **int** *numlines*, *numcols*, *numchars*, *length*, *mode*;
> **bool** *boolf*;
> **WINDOW** **win*, **win1*, **win2*, **oldwin*, **newwin*;
> **PANE** **pane*;
> **PANEL** **panel*;

Return Values

Unless otherwise noted, each routine returns a value of type **int** that is either **OK** (indicating successful completion) or **ERR** (if an error is encountered).

The Extended Curses Routines

The Extended Curses routines are listed here alphabetically, except that routines with **w**, **p**, and **mv** prefixes are listed with the corresponding routine that does not have these prefixes.

addch (*xc***)**
waddch (*win*, *xc***)**
paddch (*pane*, *xc***)**
mvaddch (*line*, *col*, *xc***)**
mvwaddch (*win*, *line*, *col*, *xc***)**
mvpaddch (*pane*, *xc***)**

> The *xc* parameter is a value of type **NLSCHAR**, rather than a single-byte **char** as used by **curses**.
>
> The **addch** routine adds the **NLSCHAR** specified by the *xc* parameter on the window at the current (*line*, *col*) coordinates. **paddch** adds the character to the presentation space for the pane specified by the *pane* parameter. If the character is `'\n'` (new-line character), the line is cleared to the end, and the current (*line*, *col*) coordinates will be changed to the beginning of the next line. A `'\r'` (return character) moves the current position to the beginning of the current line on the window. A `'\t'` (tab character) is expanded into spaces in the normal tabstop positions of every eighth column.
>
> Adding a character to the lower right corner of a window that includes the lower right corner of the display causes many terminals to scroll the entire display image

up one line. If adding a character or a character attribute causes such scrolling to occur, then **addch** makes the change on the window, but does not mark it for **wrefresh** purposes; **addch** returns the value **ERR**. Adding a single-shift control by itself to the window does not change the current position in the window.

If the current position in the window contains only a single-shift control code and *xc* is a valid character data code, then the two are combined to form an extended character, and it is added to the window at the current position. Otherwise, *xc* is treated as a valid **NLSCHAR** and is added to the window at the current position.

addstr (*string***)**
waddstr (*win*, *string***)**
paddstr (*pane*, *string***)**
mvaddstr (*line*, *col*, *string***)**
mvwaddstr (*win*, *line*, *col*, *string***)**
mvpaddstr (*pane*, *line*, *col*, *string***)**

The **addstr** routine adds the string pointed to by the *string* parameter on the window at the current (*line*, *col*) coordinates. The string can contain single-shift control codes.

Upon successful completion, **addstr** returns **OK** and the current (*line*, *col*) coordinates point to the location just beyond the end of the string. The **addstr** routine returns **ERR** if an attempt is made to add a character to the lower right corner of a window that includes the lower right corner of the display. In this case, **addstr** writes as much of the string on the window as possible.

waddfld (*win*, *string*, *length*, *numlines*, *numcols*, *mode*, *xc***)**

The **waddfld** routine adds data to a field within a window. The current coordinates specify the upper-left corner of the field in the window. The *numlines* and *numcols* parameters specify the number of lines and columns in the field, respectively. The *length* parameter specifies the length of the data. The *mode* parameter specifies the attribute for the field output. The *xc* parameter specifies the **NLSCHAR** that is used to fill the remainder of the field after the data has been added to it.

If the string contains a `'\n'` (new-line character), then the fill character is added to the reminder of the columns on that line of the field, and the remainder of the data is added starting at the first column of the next line of the field. A `'\r'` (return character) changes the current position to the beginning column of the field. A `'\t'` (tab character) is expanded with fill characters up to the next normal tabstop position within the field.

The **waddfld** routine follows the same rules as **addch** for adding single-shift control codes and character data codes to the window.

beep ( )

The **beep** routine sounds the speaker or bell at the work station.

box (*win*, *vert*, *hor*)
NLSCHAR *vert*, *hor*;

The **box** routine draws a box around the window specified by the *win* parameter. **box** uses the **NLSCHAR** specified by the *vert* parameter to draw the vertical sides of the box, and the **NLSCHAR** specified by the *hor* parameter for drawing the horizontal lines and corners.

If the window includes the lower right corner of the display and **scrollok** is not set, then the lower right corner of the box is not shown on the window and the **box** routine returns **ERR**.

The **box** routine is a macro that invokes **superbox**.

cbox (*win*)

The **cbox** routine draws a box around the window specified by the *win* parameter. The characters used are those defined in **/usr/lib/terminfo** (type 1 box characters) or defaulted during the initialization.

The **cbox** routine is implemented as a macro that invokes **superbox**.

The **cbox** routine returns **ERR** if the window includes the lower right corner of the display and **scrollok** is not set on.

chgat (*numchars*, *mode*)
wchgat (*win*, *numchars*, *mode*)
pchgat (*pane*, *numchars*, *mode*)
mvchgat (*line*, *col*, *numchars*, *mode*)
mvwchgat (*win*, *line*, *col*, *numchars*, *mode*)
mvpchgat (*pane*, *line*, *col*, *numchars*, *mode*)

The **chgat** routine changes the attributes of the next *numchars* characters on the window starting from the current (*line*, *col*) coordinates. The attributes are changed to the attributes specified by the *mode* parameter. This routine will not wrap around to the next line; however, specifying a value for the *numchars* parameter that would cause a line wrap is not an error.

The *mode* parameter is one or more of the attributes defined by the global attribute variables. More than one attribute may be specified by logically OR-ing them together. The following example changes the attributes of the next 10 characters to bold blue characters on a black background:

```
chgat (10, BOLD | F_BLUE | B_BLACK)
```

The **chgat** routine returns **ERR** if the change forces scrolling and **scrollok** is not set on for the window.

clear ( )
wclear (*win*)

The **clear** routine resets the entire **stdscr** window to blank characters. **clear** sets the current (*line*, *col*) coordinates to (0, 0).

clearok (*scr*, *boolf*)
WINDOW *scr;

The **clearok** routine sets the clear flag for the screen specified by the *scr* parameter. If the *boolf* parameter is **TRUE**, then the screen will be cleared on the next call to **refresh** or **wrefresh**. If the *boolf* parameter is **FALSE**, then the screen will not be cleared on the next call to **refresh** or **wrefresh**. This only works on screens, and, unlike **clear**, does not alter the contents of the screen. If the *scr* parameter is **curscr**, then the next **refresh** will cause a clear-screen, even if the window passed to **refresh** is not a screen.

The **clearok** routine returns **ERR** if the window is not a full screen window.

clrtobot ( )
wclrtobot (*win*)

The **clrtobot** routine erases the window from the current (*line*, *col*) coordinates to the bottom. **clrtobot** leaves the current (*line*, *col*) coordinates unchanged. This does not force a clear-screen sequence on the next refresh.

The **clrtobot** routine always returns the value **OK**.

clrtoeol ( )
wclrtoeol (*win*)

The **clrtoeol** routine clears the window from the current (*line*, *col*) coordinates to the end of the current line. The current (*line*, *col*) coordinates are not changed.

The **clrtoeol** routine always returns the value **OK**.

colorend ( )
wcolorend (*win*)

The **colorend** routine returns the terminal to **NORMAL** mode. By default, **NORMAL** is usually defined as **(F_WHITE | B_BLACK)**.

The **colorend** routine is a macro that invokes **xstandend**.

The **colorend** routine always returns the value **OK**.

colorout (*mode*)
wcolorout (*win*, *mode*)

The **colorout** routine sets the current standout bit-pattern of the window (*win*->**_csbp**) to the attribute specified by the *mode* parameter. Characters added to the window after such a call will have *mode* as their attribute. The *mode* parameter is constructed by logically OR-ing together attributes that are declared in the **cur01.h** header file that are supported by the terminal.

The **colorout** routine overrides the current setting of the window, and will work in conjunction with almost all of the routines that cause output to be placed on the window.

The **colorout** routine is a macro that invokes **xstandout**.

The **colorout** routine always returns the value **OK**.

cresetty (*boolf*)

The **cresetty** routine resets the terminal to the state saved by the last call to **csavetty**. Use this routine after the completion of a program that uses the terminal as a simple terminal. If the *boolf* parameter is **TRUE**, then the data in **curscr** is redisplayed.

crmode ( )
nocrmode ( )

The **crmode** routine turns off the ***canonical processing*** of input by the system device driver. When canonical processing is off, data is made available without waiting for a `'\n'` (new-line character). **nocrmode** enables canonical processing by the system device driver.

The **wgetch** routine, which is used for all Extended Curses input, forces the equivalent of **crmode** before requesting input if echoing is active, and reinstates the original status on exit. If you are using echo, you should issue a call to either **crmode** or **raw** to avoid multiple calls by **wgetch**.

The **crmode** routine differs from **raw** in that **crmode** has no effect on output data processing and does not disable signal processing by the device driver.

The **crmode** routine always returns the value **OK**.

csavetty (*boolf*)

The **csavetty** routine saves the current Extended Curses state so that it can later be reset by **cresetty**. Use this routine before running a program that uses the terminal as a simple terminal. If the *boolf* parameter is **TRUE**, then the following status is set before saving the terminal status: **crmode**, **noecho**, **meta**, **nonl**, and **keypad (TRUE)**.

delay

See "nodelay" on page 3-156.

delch ( )
wdelch (*win*)
mvdelch (*line*, *col*)
mvwdelch (*win*, *line*, *col*)

The **delch** routine deletes the character at the current (*line*, *col*) coordinates. Each character after the deleted character on the line shifts to the left, and the last character becomes blank.

The **delch** routine always returns the value **OK**.

deleteln ( )
wdeleteln (*win*)

The **deleteln** routine deletes the current line. Every line below the current line moves up, and the bottom line becomes blank. The current (*line*, *col*) coordinates remain unchanged.

The **deleteln** routine always returns the value **OK**.

delwin (*win*)

The **delwin** routine deletes the window specified by the *win* parameter. All resources used by the deleted window are freed for future use.

If a window has a subwindow allocated inside of it, the deletion of the window does not affect the subwindow even though the subwindow is invalidated. Therefore, subwindows must be deleted before the outer windows are deleted.

The **delwin** routine always returns the value **OK**.

drawbox (*win*, *line*, *col*, *numlines*, *numcols*)

The **drawbox** routine draws a box with the upper left corner located at the position specified by the *line* and *col* parameters. The *numlines* parameter specifies the number of rows to be used by the box, and the *numcols* parameter specifies the number of columns to be used by the box.

The characters used to draw the box are either those specified in the **terminfo** file, or those defaulted at initialization.

The **drawbox** routine returns **ERR** if part or all of the box is outside the window, or the box addresses the lower right corner of the screen and **scrollok** is not on.

#include <cur05.h>

ecactp (*pane*, *boolf*)

The **ecactp** routine specifies the active pane in a panel. The pane specified by the *pane* parameter is made the active pane if the *boolf* parameter is **TRUE**. If an active pane has been previously designated, then the border of that pane is reset to the inactive display mode, and the border of the pane specified by the *pane* parameter is set to the active display mode. If the *boolf* parameter is **FALSE**, then the border of the pane specified by the *pane* parameter is set to the inactive display mode.

#include <cur05.h>

ecadpn (*pane*, *win*)

The **ecadpn** routine adds the window specified by the *win* parameter to the list of windows that can be presented in the pane specified by the *pane* parameter. No visible action occurs as a result of this routine. A call to **ecaspn** must be made after **ecadpn** to change the data associated with the pane display.

The **ecadpn** routine returns **ERR** if the system is unable to allocate the storage required.

#include <cur05.h>

ecaspn (*pane*, *win*)

The **ecaspn** routine makes the window specified by the *win* parameter the current window for display in the pane specified by the *pane* parameter. A refresh call for the pane or panel is needed to cause the data to be presented on the display. The viewport associated with the pane is positioned with the top left corner of the viewport at the top left corner of the data for the window.

The **ecaspn** routine returns **ERR** if the window specified by the *win* parameter was not previously associated with this pane using **ecadpn**.

WINDOW *ecblks ( )

The **ecblks** routine returns a pointer to a window that is filled with blanks. This window is intended to be used as a filler for panes that have no real content. It requires less storage than normal windows because all lines will always contain blanks.

Do not modify or delete this window.

#include <cur05.h>

PANEL *ecbpls (*numlines*, *numcols*, *firstline*, *firstcol*, *title*, *divdim*, *border*, *pane***)**
short *numlines*, *numcols*, *firstline*, *firstcol*;
char **title*;
char *divdim*, *border*;

The **ecbpls** routine builds a panel structure.

The *numlines* parameter specifies the panel size in rows.

The *numcols* parameter specifies the panel size in columns.

The *firstline* parameter specifies the panel's origin on the display's upper left corner row coordinate.

The *firstcol* parameter specifies the panel's origin on the display's upper left corner column coordinate.

The *title* parameter points to a title string. The title is shown centered in the top border. If no title is desired, this parameter should be **NULL**.

The *divdim* parameter specifies the dimension along which this panel is to be divided: either **Pdivtyv** (vertical) or **Pdivtyh** (horizontal).

The *border* parameter indicates whether or not this panel is to have a border: either **Pbordry** (yes) or **Pbordrn** (no).

The *pane* parameter points to the first pane that defines the divisions of this panel.

All parameters should be given as defined here. However, they are not checked or used until a call is made to **ecdvpl**. An application may modify values put into this structure until it calls **ecdvpl**.

Upon successful completion, a pointer to the new panel is returned. **ecbpls** returns **ERR** if there is not enough storage available.

#include <cur05.h>

PANE *ecbpns (*numlines*, *numcols*, *ln*, *ld*, *divdim*, *ds*, *du*, *border*, *lh*, *lv***)**
short *numlines*, *numcols*, *ds*;
PANE **ln*, **ld*, **lh*, **lv*;
char *divdim*, *du*, *border*;

The **ecbpns** routine builds a pane structure.

The *numlines* parameter specifies the number of rows in the presentation space for the pane.

The *numcols* parameter specifies the number of columns in the presentation space for the pane.

The *ln* parameter points to a neighboring pane either above or to the left.

The *ld* parameter points to the start of a chain for divisions of the pane.

The *divdim* parameter specifies the dimension of the pane along which division is to occur. This parameter is used if and only if the *ld* parameter is not **NULL**. Valid values for this parameter are **Pdivpnv** (vertical dimension) and **Pdivpnh** (horizontal dimension).

The *ds* and *du* parameters together specify the size of this pane as part of the division of a parent pane:

du	**Vertical or Horizontal Size of the Pane**
Pdivszc	The size is specified by the *ds* parameter.
Pdivszp	The size is *ds* ÷ 10000 of the available space. For example, if *ds* is 5000, then the row or column size is half of the available space.
Pdivszf	The pane has a floating size. The value of the *ds* parameter is not used.

The *border* parameter specifies whether or not this pane has a border: either **Pbordry** (yes) or **Pbordrn** (no).

The *lh* parameter points to a pane that is to scroll with this pane when the pane scrolls horizontally.

The *lv* parameter points to a pane that is to scroll with this pane when the pane scrolls vertically.

If you specify **NULL** for the *ld* parameter, or if you are not sure which value to use for *du*, then specify **Pdivszf** for the *du* parameter.

If the *ln* parameter is not **NULL**, the **divs** field of the pane structure being built receives the value that was in the *ln*.**divs** field. The *ln*.**divs** field is modified to point to the new pane structure being built.

If the *lh* and the *lv* parameters are not **NULL**, they will be used to link the new structure to the specified structures and to link the specified structures to the new structure. The links thus created will form a ring that includes all panes that scroll together.

Upon successful completion, a pointer to the new pane structure is returned. **ecbpns** returns **ERR** if a error is detected during processing.

#include <cur05.h>

ecdfpl (*panel*, *boolf*)

The **ecdfpl** routine creates the Extended Curses **WINDOW** structures needed to define the specified panel.

At the time this routine is invoked, all size and location specifications of the panel and its constituent panes must be properly set. **ecdfpl** does not examine any of the division size specifications or the scroll link specifications.

The **fpane** pointer in the indicated **PANEL** structure must point to the first leaf pane for the panel, and the subsequent **nxtpn** pointers from that pane must form a loop back to the first leaf pane. (This is done by **ecdvpl**.)

A **WINDOW** structure is built for the panel specified by the *panel* parameter. This **WINDOW** will have a size that corresponds to the size of the panel. For each of the panes in the subsequent chain, a separate **WINDOW** structure is built with a size that corresponds to the specified presentation space size or the viewport size, whichever is larger.

If borders are specified for any of the panes, those borders are drawn on the **WINDOW** for the panel. All corners are checked and, if needed, proper junction characters are used to draw the corner.

The *boolf* parameter indicates whether to suppress the creation of presentation spaces for the panes. If the value is **TRUE**, then presentation spaces are not created. If **FALSE**, then presentation spaces are created.

The **ecdfpl** routine returns **ERR** if sufficient storage is not available for the **WINDOW** structures being created.

#include <cur05.h>

ecdppn (*pane*, *oldwin*, *newwin*)

The **ecdppn** routine adds, drops or replaces a presentation space for a pane.

First, if the *oldwin* parameter is not **NULL**, then **ecdppn** drops *oldwin* from the list of windows that are alternatives for the pane specified by the *pane* parameter. The previous association should have been established using **edadpn**. If the *oldwin* parameter is **NULL**, then no window is dropped.

Next, if the *newwin* parameter is not **NULL**, then **ecdppn** adds *newwin* as a valid pane for this window, replacing *oldwin*, if it was associated with the pane specified by the *pane* parameter. (See **ecadpn** for a better way to add a pane).

The **ecdppn** routine always returns the value **OK**.

#include <cur05.h>

ecdspl (*panel*)

The **ecdspl** routine releases all of the data structures associated with the panel specified by the *panel* parameter. The released data structures are returned to the free pool. The released data structures include the panel structure, all associated pane structures, any window structures associated with the panes, any auxiliary window structures associated with the panes, and all private control structures used by Extended Curses.

#include <cur05.h>

ecdvpl (*panel*)

The **ecdvpl** routine assigns a real size and relative position to all the panes defined for the panel specified by the *panel* parameter. All of the panes must be linked to the panel. The structure of a tree will be followed to determine the sizes for each pane.

The direction of the first set of divisions and the size of the first set of divisions is determined. This information is used to control the division algorithm. Using the size along the direction of division, first, the total space for the interior of panes is determined by counting the panes and their borders. Next, any panes with *fixed* size are given the space indicated by the **divsz** field in the pane structure. The remaining available space is then assigned to the panes that have specified a *proportional* size. Finally, any space that remains is assigned to those panes that specified a *floating* size. Once the sizes are determined, the origin for each pane relative to the panel origin is determined and entered into the PANE structure. A final pass is made over the list of panes in the current division, and, for each that is itself divided, the process is repeated.

If adjacent panes both have a border specified, the border space is shared between them.

If all of the panes have a fixed size and the total is less than the available space, there will be space that cannot be accessed by the application in the resulting structure.

If, after allocating space to the proportional panes, there is space remaining and no floating panes are in the current set, the remaining free space is allocated to the proportional panes.

The **ecdvpl** routine returns **ERR** and the structures are invalid for use by **ecdfpl** if one or more of the following occur:

- The total size specified for fixed panes exceeds the space available.
- The total fractions specified for the proportional panes exceed a total of 1.
- The number of panes exceeds the number of positions available.

#include <cur05.h>

ecflin (*pane*, *firstline*, *firstcol*, *numlines*, *numcols*, *pat*, *xc*, *buf*, *mask*)

NLecflin (*pane*, *firstline*, *firstcol*, *numlines*, *numcols*, *pat*, *xc*, *buf*, *length*, *mask*)
char **pat*, **buf*, **mask*;

The **ecflin** and **NLecflin** routines input field data to a pane. **NLecflin** is supplied for international character support, and **ecflin** is retained to preserve traditional functionality. **NLecflin** works like **ecflin**, but has an additional parameter, *length*, which specifies the length of the buffer in which the input data is stored.

The **ecflin** routine inputs field data to the pane pointed to by the *pane* parameter. The *firstline* and the *firstcol* parameters specify the upper left corner of the field in the current window being shown in the pane. The *numcols* parameter specifies the number of columns in the field, and the *numlines* parameter specifies the number of rows in the field.

The *buf* parameter points to a buffer into which input data is stored. The buffer must be at least *numlines* × *numcols* characters long.

The *xc* parameter specifies the first **NLSCHAR** to be entered into the field. If the *xc* parameter is a null character, it is ignored.

The *pat* and *mask* parameters specify the set of characters that are to be accepted as valid input.

The position in the field may not always correspond to the position in the input buffer, since a 2-byte extended character corresponds to a single display character in the field in the window. Input is accepted from the terminal as long as the cursor remains within the bounds of the field. However, if the input buffer is filled before the cursor exits the field, input processing stops and **ecflin** returns.

Cursor movement that moves the cursor outside the field is allowed and is reflected on the display. If cursor movement places the cursor in a position where data input would cause the input buffer to overflow, input processing stops. Any data keys entered are checked against the character set specified by the *pat* parameter. If the data character is acceptable, then it is echoed. If the character is not acceptable, then the **ecflin** routine returns its value.

Insert and delete keys are honored and data is shifted within the field as needed. If the field spans more than one line and insertions or deletions are made, then data that is shifted out of one line of the field is shifted into the end of the next line. Data shifted out of the field is lost. When characters are deleted, null characters are shifted into the end of the field.

The *pat* parameter points to a string that indicates the set of characters that are acceptable as valid input. These characters include all code points of the P0, P1, and P2 code pages (see "display symbols" on page 5-24). The string is formed of these codes:

U Uppercase letters: 'A'—'Z' plus the accented uppercase letters from code pages P0, P1, and P2.

L Lowercase letters: 'a'—'z' plus the accented lowercase letters from code pages P0, P1, and P2.

N Numeric characters: '0'—'9'.

A Alphanumeric characters: 'A'—'Z', 'a'—'z', and '0'—'9' plus the accented letters from code pages P0, P1, and P2.

B Blank (space character—0x20).

P Printable characters: *blank*—'~' (0x20—0x7E).

G Graphic characters: '!'—'~' (0x21—0x7E).

X Hexadecimal characters: '0'—'9', 'A'—'F', and 'a'—'f'.

C Control Characters:

- Cursor Up, Cursor Down, Cursor Left, Cursor Right
- Backspace
- Back-tab (to first position of field)
- Insert (enable or disable insert mode)
- Delete (delete current character)
- New-line (to left column and down one line)

D Default characters:

- 0x20—0x7E
- 0x80—0xFF
- 0x1FA0—0x1FFF
- 0x1E80—0x1EFF
- 0x1DA0—0x1DFF
- 0x1C80—0x1CFF
- Controls, as defined for code C.

Z Application-specified character set

+ Allows characters indicated by following codes.

- Does not allow characters indicated by following codes.

If the first character of *pat* is + or -, then the set of characters specified by the rest of the string is added to (+) or taken from (-) the default characters (which can also be specified with D). If the first character in this string is not + or -, then the set of characters specified by *pat* replaces the default. After the first character, the sets indicated are allowed unless preceded by a - (minus sign). For example:

"PC-L" Allows the printable and control characters, except for lowercase letters.

"-CBN" Allows all of the default characters, except for control characters, blanks, or numeric characters.

If the *pat* string contains a Z, then the array pointed to by the *mask* parameter specifies a character validity mask. This array must be exactly 128 bytes long (1024 bits), where each bit corresponds to a character code as returned by **wgetch**. The bytes in the array correspond as follows:

Bytes 0—31	P0 characters 0x00—0xFF
Bytes 32—63	Keycodes 0x100—0x1FF
Bytes 64—79	Low P1 characters 0x1F80—0x1FFF
Bytes 80—95	High P1 characters 0x1E80—0x1EFF
Bytes 96—111	Low P2 characters 0x1D80—0x1DFF
Bytes 112—127	High P2 characters 0x1C80—0x1CFF

If a given bit is set to 1, then the corresponding character is accepted (for +Z) or rejected (for -Z). If a bit is set to 0, then the acceptance status of the corresponding character, as determined by the rest of *pat*, is not changed.

Upon successful completion, the code associated with the last input that terminated input is returned.

The **ecflin** routine returns **ERR** if one or more of the following are true:

- There is an error in the parameters.
- The *firstline* parameter is outside the window.
- The *firstcol* parameter is outside the window.
- The *numcols* parameter is too large.
- The *numlines* parameter is too large.

echo ( )
noecho ( )

The **echo** routine causes the terminal to echo characters to the display. If **echo** is set on, **wgetch** places all input into the data structure for the window.

The **noecho** routine turns **echo** off. If **echo** is turned off, characters are not written to the display.

#include <cur05.h>

ecpnin (*pane*, *boolf*, *xc*)

The **ecpnin** routine causes the pane to accept keyboard input. The pane specified by the *pane* parameter is scrolled, if necessary, to insure that the cursor is visible on the display. Keyboard input is then accepted. If the *boolf* parameter is **TRUE** and if the input character is a simple cursor movement, then the resulting cursor position is reflected on the display. Further input is then read from the terminal. If the *boolf* parameter is **FALSE**, or if the input character is not a simple cursor movement, then the value of the input character is returned.

The *xc* parameter specifies the first **NLSCHAR** to be assumed from the display. If *xc* is a null character, then it is ignored.

This routine tracks the locator cursor if locator tracking is enabled (see "trackloc" on page 3-161).

void ecpnmodf (*pane*)

The **ecpnmodf** macro marks the panel that contains the pane specified by the *pane* parameter as modifed. This information is used by **ecrfpl** to determine whether a panel needs to be written to the display.

#include <cur05.h>

ecrfpl (*panel*)

The **ecrfpl** routine refreshes the panel specified by the *panel* parameter. If that panel is partially obscured by other panels, then those panels are also written to the display. If the *panel* parameter is **NULL**, then all panels that have been marked as modified (with **ecpnmodf**) are written. If any panels have been removed (with **ecrmpl**), then all panels are written.

#include <cur05.h>

ecrfpn (*pane*)

The **ecrfpn** routine refreshes the pane specified by the *pane* parameter on the display. If the pane is the active pane, then the window might be scrolled to assure that the cursor is visible. If the pane is not active, then the window is not scrolled.

The **ercfpn** routine always returns the value **OK**.

#include <cur05.h>

ecrlpl (*panel*)

The **ecrlpl** routine returns the structures associated with the panel specified by the *panel* parameter to the free storage pool. This includes all window structures associated with the panes of the panel, all Extended Curses private structures, and any added window structures. The panel and associated pane structures are not released and can be reused.

The **ecrlpl** routine always returns the value **OK**.

#include <cur05.h>

ecrmpl (*panel*)

The **ecrmpl** routine removes the panel specified by the *panel* parameter from the list of panels that are currently being displayed. If the panel is not currently in that list, no action is taken and no error is returned. This routine should be followed by a call to **ecrfpl** to update the display.

The **ecrmpl** routine always returns the value **OK.**

#include <cur05.h>

ecscpn (*pane*, *numlines*, *numcols*)

The **ecscpn** routine causes the pane specified by the *pane* parameter to be scrolled over the underlying window the distance indicated by the *numcols* and the *numlines* parameters. The *numcols* parameter specifies the distance to scroll horizontally and the *numlines* parameter specifies the distance to scroll vertically. These parameters can be positive or negative and may imply a movement that positions the viewport partially or completely off the window. If such a position results from the scroll, the scroll stops after moving as far in the indicated direction as possible. Positive values move to the right or down. Negative values move to the left or up.

If there are other panes linked to the pane specified, those panes will also scroll an amount necessary to maintain the identical horizontal or vertical positioning on the respective windows. If the resulting position requires placing the viewport partially or completely off the window, the scroll request terminates at the edge of the window.

#include <cur05.h>

ecshpl (*panel*)

The **ecshpl** routine shows the panel specified by the *panel* parameter on the terminal.

If the specified panel is currently the top panel, no action is taken and no error is returned. If there is another top panel, the active pane in that panel is changed to the inactive state. The specified panel is placed at the top of the panel chain. This routine should be followed by a call to **ecrfpl** to update the display.

The **ecshpl** routine always returns the value **OK.**

#include <cur05.h>

ectitl (*title*, *line*, *col*)
**char ****title*;

The **ectitl** routine creates or modifies the title panel. The title panel is always visible, that is, on top of any other panels. The *title* parameter points to a character string that is displayed as the new title. If *title* is **NULL**, then any existing title is removed. The *line* and *col* parameters specify the coordinates for the upper left corner of the title panel. If *firstline* is not valid, then it defaults to 1. If *firstcol* is not valid, then the title will be centered.

endwin ( )

The **endwin** routine ends window routines before exiting. Ending window routines before exiting restores the terminal to the state it was before **initscr** (or **gettmode** and **setterm**) was called. **endwin** should always be called before exiting. **endwin** does not exit.

erase ( )
werase (*win*)
perase (*pane*)

The **erase** routine clears the window and sets it to blanks without setting the clear flag. Similarly, **perase** erases the pane specified by the *pane* parameter. This is analogous to the **clear** routine, except that it does not cause a clear-screen sequence to be generated on a **refresh**.

extended (*boolf*)

The **extended** routine turns on and off the combining of input bytes into 2-byte extended characters. If the *boolf* parameter is **TRUE**, then this input processing is turned on; if **FALSE**, then it is turned off. By default, **extended** processing is initially turned on.

flash ( )

The **flash** routine displays a visual "bell" on the terminal screen if one is available. If a visual bell is not available, then **flash** toggles the terminal speaker or bell.

The **flash** routine always returns the value **OK**.

fullbox (*win, vert, hor, topl, topr, botl, botr***)**
NLSCHAR *vert, hor, topl, topr, botl, botr*;

The **fullbox** routine puts box characters on the edges of the window. The *vert* parameter specifies the **NLSCHAR** to use for the vertical sides. The *hor* parameter specifies the **NLSCHAR** to use for the horizontal lines. The *topl* and the *topr* parameters specify the **NLSCHARs** to use for the top left and the top right corners. The *botl* and the *botr* parameters specify the **NLSCHARs** to use for the bottom left and the bottom right corners.

The **fullbox** routine returns **ERR** if an attempt is made to scroll when **scrollok** is not active.

The **fullbox** routine is a macro that invokes **superbox**.

#include <cur02.h>

int getch ( )
int wgetch (*win***)**
int mvgetch (*line, col***)**
int mvwgetch (*win, line, col***)**

The **getch** routine gets a character from the terminal and echoes it on the window, if necessary. If **noecho** has been set, then the window does not change. **noecho** and either **crmode** or **raw** must be set for Extended Curses to know what is actually on the terminal. If these settings are not correct, **wgetch** sets **noecho** and **crmode** and resets them to the original mode when done.

If **extended** processing is turned on, then **getch** combines input sequences that contain single-shift controls into 2-byte extended characters. By default, **extended** processing is turned on. (See "extended" on page 3-150.)

Upon completion, the character code for the data key or one of the following values is returned:

KEY_NOKEY	**nodelay** is active and no data is available.
KEY_*xxxx*	**keypad** is active and a control key was recognized. See the **cur02.h** header file for a complete list of the key codes that can be returned.
ERR	Echoing the character would cause the screen to scroll illegally.

#include <cur02.h>

int getstr (*string*)
int wgetstr (*win*, *string*)
int mvgetstr (*line*, *col*, *string*)
int mvwgetstr (*win*, *line*, *col*, *string*)

The **getstr** routine gets a string through the window and stores it in the location pointed to by the *string* parameter. The string may contain single-shift control codes. The area pointed to must be large enough to hold the string. **getstr** calls **wgetch** to get the characters until a new-line character or some other control character is encountered.

Upon completion, one of the following values is returned:

OK	The input string was terminated with a new-line character.
KEY_NOKEY	**nodelay** is active and no data is available.
KEY_*xxxx*	The input string ended with a control key, and the code for this key was returned. See the **cur02.h** header file for a complete list of the key codes that can be returned.
ERR	The string caused the screen to scroll illegally.

gettmode ( )

The **gettmode** routine issues the needed control operation to the display device driver to save the processing flags in a fixed global area. **gettmode** is invoked by **initscr** and is not normally called directly by applications.

getyx (*win*, *line*, *col*)

The **getyx** routine stores the current (*line*, *col*) coordinates of window specified by the *win* parameter into the variables *line* and *col.* Because **getyx** is a macro and not a subroutine, the names of *line* and *col* passed, not their addresses.

Upon successful completion, *line* and *col* contain the current row and column coordinates for the cursor in the specified window.

NLSCHAR inch ( )
NLSCHAR winch (*win*)
NLSCHAR mvinch (*line*, *col*)
NLSCHAR mvwinch (*win*, *line*, *col*)

The **inch** routine returns the **NLSCHAR** at the current (*line*, *col*) coordinates on the specified window. No changes are made to the window.

Upon successful completion, the code for the character located at the current cursor location is returned.

WINDOW *initscr ( )

The **initscr** routine performs screen initialization. **initscr** must be called before any of the screen routines are used. It initializes the terminal-type data, and without it, none of the Extended Curses routines can operate properly.

If standard input is not a **tty**, **initscr** sets the specifications to the terminal whose name is pointed to by **Def_term** (initially "dumb"). If the value of the **bool** global variable **My_term** is **TRUE**, **Def_term** is always used.

If standard input is a terminal, the specifications for the terminal named in the environment variable **TERM** are used. These specifications are obtained from the **terminfo** description file for that terminal.

The **initscr** routine creates the structures for **stdscr** and **curscr** and saves the pointers to those structures in global variables with the corresponding names.

Upon successful completion, a pointer to **stdscr** is returned.

insch (*xc*)
winsch (*win*, *xc*)
mvwinsch (*win*, *line*, *col*, *xc*)
mvinsch (*line*, *col*, *xc*)

The **insch** routine inserts the **NLSCHAR** specified by the *xc* parameter into the window at the current (*line*, *col*) coordinates. Each character after the inserted character shifts to the right and the last byte on the line disappears.

The **insch** routine always returns the value **OK**.

insertln ( )
winsertln (*win*)

The **insertln** routine inserts a line above the current line. Each line below the current line is shifted down, and the bottom line disappears. The current line becomes blank and the current (*line*, *col*) coordinates remain unchanged.

The **insertln** routine always returns the value **OK**.

keypad (*boolf*)

The **keypad** routine turns on and off the mapping of key sequences to single integers. If the *boolf* parameter is **TRUE**, input processing is turned on. If the *boolf* parameter is **FALSE**, input processing is turned off. By default, input processing is initially turned off.

When turned on, sequences of characters from the terminal are translated into integers that are defined in the **cur02.h** header file.

The codes available on a given terminal are determined by the **terminfo** terminal description file.

The **keypad** routine always returns the value **OK**.

leaveok (*win*, *boolf*)

The **leaveok** routine sets a flag, used by the window specified by the *win* parameter, which controls where the cursor is placed after the window is refreshed. If the *boolf* parameter is **TRUE**, when the window is refreshed the cursor is left at the last point where a change was made on the terminal, and the current (*line*, *col*) coordinates for the window specified by the *win* parameter are changed accordingly. If the (*line*, *col*) coordinates are outside the window, the coordinates are forced to (0, 0). If the *boolf* parameter is **FALSE**, when the window is refreshed the cursor is moved to the current (*line*, *col*) coordinates within the window. The controlling flag is initially set to **FALSE**.

The **leaveok** routine always returns the value **OK**.

char *longname ( )

The **longname** routine returns a pointer to a static area that contains the long (full) name of the terminal as it appears in the **terminfo** entry for the terminal.

meta ( )
nometa ( )

The **meta** routine prevents the stripping of the eighth bit of each keyed character.

The **nometa** routine causes the eighth or most-significant bit of each keyed character to be stripped. Not all terminals support the stripping of bits.

The **meta** and **nometa** routines always return the value **OK**.

move (*line*, *col*)
wmove (*win*, *line*, *col*)

The **move** routine changes the current (*line*, *col*) coordinates of the window to the coordinates specified by the *line* and *col* parameters.

The **move** routine returns **ERR** if the destination for the cursor is outside the window or viewport.

mvcur (*line*, *col*, *newline*, *newcol*)
int *line*, *col*, *newline*, *newcol*;

The **mvcur** routine moves the terminal's cursor from the coordinates specified by the *line* and *col* parameters to the coordinates specified by the *newline* and *newcol* parameters.

It is possible to use this optimization without the benefit of the screen routines. In fact, **mvcur** should not be used with the screen routines. Use **move** and **refresh** to move the cursor position and inform the screen routines of the move.

mvwin (*win*, *line*, *col*)

The **mvwin** routine moves the position of the viewport or the subwindow specified by the *win* parameter from its current starting coordinates to the coordinates specified by the *line* and *col* parameters. The *line* parameter specifies the row on the display for the top row of the window. The *col* parameter specifies the column on the display for the first column of the window.

The **mvwin** routine returns **ERR** if a part of the window position is outside the bounds of the window on which the viewport is defined.

WINDOW ***newview** (*win*, *numlines*, *numcols*)

The **newview** routine creates a new window that has the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new window is a viewport of the window specified by the *win* parameter and starts at the current (*line*, *col*) coordinates of the window specified by the *win* parameter. The resulting window's initial position on the display is set to (0, 0).

The viewport window returned by **newview** is a special subwindow that is suitable for viewport scrolling. Viewport scrolling here refers to the type of scrolling that is characteristic of full-screen editors.

Because the returned viewport window is a subwindow, any change made in either window in the area covered by the viewport window appears in both windows. Both windows actually share the relevant storage area. A viewport window cannot be scrolled using **scroll**.

Other than the exceptions noted above, viewport windows behave like subwindows.

Upon successful completion, a pointer to the control structure for the new viewport is returned.

The **newview** routine returns **ERR** if the window specified by the *win* parameter is a subwindow or a viewport, or if sufficient storage is not available for the new structures.

WINDOW *newwin (*numlines*, *numcols*, *firstline*, *firstcol*)

The **newwin** routine creates a new window that contains the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new window will start at the coordinates specified by the *firstline* and the *firstcol* parameters.

If the *numlines* parameter is 0, then that dimension is set to (**LINES** - *firstline*). If the *numcols* parameter is 0, then that dimension is set to (**COLS** - *firstcol*). Therefore, to get a new window of dimensions (**LINES** × **COLS**), use:

```
newwin (0, 0, 0, 0)
```

The size specified for the window can exceed the size of the real display. In this case, a viewport or subwindow must be used to present the data from the window on the terminal.

Upon successful completion, a pointer to the new window structure is returned.

The **newwin** routine returns **ERR** if any of the parameters are invalid, or if there is insufficient storage available for the new structure.

nl ( )
nonl ( )

The **nl** routine sets the terminal to ***nl mode***. When in nl mode, the system maps `'\r'` (return characters) to `'\n'` (new-line or line-feed characters). If the mapping is not done, **refresh** can do more optimization. **nonl** turns nl mode off.

The **nl** routine and **nonl** do not affect the way in which **waddch** processes new-line characters.

The **nl** and **nonl** routines always return the value **OK**.

nodelay (*boolf*)

The **nodelay** routine controls whether read requests wait for input if no keystroke is available. If the *boolf* parameter is **FALSE**, then the read routines wait for operator input. This is the default setting. If the *boolf* parameter is **TRUE**, then the read routines return immediately if no keyboard data is available.

If **nodelay** is set (**TRUE**) and if no keystroke is available from the keyboard, then **getch** returns **KEY_NOKEY**, defined in the **cur02.h** header file.

The **nodelay** routine always returns the value **OK**.

overlay (*win1*, *win2*)

The **overlay** routine overlays the window specified by the *win1* parameter on the window specified by the *win2* parameter. The contents of the window specified by the *win1* parameter, insofar as they fit, are placed on the window specified by the *win2* parameter at their starting (*line*, *col*) coordinates. This is done nondestructively; that is, blanks on the *win1* window leave the contents of the space on the *win2* window untouched.

The **overlay** routine moves data only if the data is non-blank or if the display attribute is different.

The only data that is considered for moving from the *win1* window to the *win2* window is data that occupies display positions that are common to both windows.

The **overlay** routine is implemented as a macro that invokes **overput** which uses **waddch** to transfer the data from window to window.

The **overlay** routine returns **ERR** if the overlay addresses the lower right corner of the display and **scrollok** is **FALSE**.

overwrite (*win1*, *win2*)

The **overwrite** routine copies data from the window specified by the *win1* parameter to the window specified by the *win2* parameter. The contents of the *win1* window, insofar as they fit, are placed on the *win2* window at their starting (*line*, *col*) coordinates. This is done destructively; that is, blanks on the *win1* window become blanks on the *win2* window.

Only the data that occupies positions on the display that are common to the two windows will be moved from the *win1* window to the *win2* window.

The **overwrite** routine is implemented as a macro that invokes **overput** which uses **waddch** to transfer the data from window to window.

The **overwrite** routine returns **ERR** if an attempt is made to write to the lower right corner and **scrollok** is not set.

printw (*fmt* [, *value*, . . .])
wprintw (*win*, *fmt* [, *value*, . . .])
char **fmt*;

The **printw** routine performs a **printf** on the window using the format control string specified by the *fmt* parameter and the values specified by the *value* parameters. The output to the window starts at the current (*line*, *col*) coordinates. Use the field width options of **printf** to avoid leaving things on the window from earlier calls. See "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300 for details.

The **printw** routine returns **ERR** if it causes the screen to scroll illegally.

raw ( )
noraw ( )

The **raw** routine sets the terminal to raw mode. In raw mode, canonical processing by the device driver and signal processing are turned off. The **noraw** routine turns off raw mode.

The **raw** and **noraw** routines always return the value **OK.**

refresh ( )
wrefresh (*win*)

The **refresh** routine synchronizes the terminal screen with the window. If the window is not a screen, then only the part of the display covered by it is updated. **refresh** checks for possible scroll errors at display time.

The **refresh** routine returns **ERR** if the change specified is in the last position of a window that includes the lower right corner of the display, or if they would cause the screen to scroll illegally. If they would cause the screen to scroll illegally, **refresh** updates whatever can be updated without causing the scroll.

resetty (*boolf*)

The **resetty** routine restores the terminal status flags that were previously saved by **savetty.** If the *boolf* parameter is **TRUE**, then the screen is cleared in addition to resetting the terminal. **resetty** is performed automatically by **endwin** and is not normally called directly by applications.

savetty ( )

The **savetty** routine saves the current terminal status flags. **savetty** is performed automatically by **initscr** and is not normally called directly by applications.

scanw (*fmt* [, *pointer*, . . .])
wscanw (*win*, *fmt* [, *pointer*, . . .])
**char ****fmt*;

The **scanw** routine performs a **scanf** through the window using the format control string specified by the *fmt* parameter. **scanw** uses **wgetstr** to obtain the string, then invokes the internal routine for **scanf** to process the data. See "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325 for details.

scroll (*win*)

The **scroll** routine moves the data in the window specified by the *win* parameter up one line and inserts a new blank line at the bottom.

scrollok (*win*, *boolf*)

The **scrollok** routine sets the scroll flag for the window specified by the *win* parameter. If the *boolf* parameter is **TRUE**, then scrolling is allowed. The default setting is **FALSE**, which prevents scrolling.

sel_attr (*set*)
int **set*;

The **sel_attr** routine allows you to change the selection and priority of attributes for the run-time terminal. The *set* parameter points to a **NULL**-terminated integer array that contains display attribute values from the **cur03.h** header file in the order that you want them regardless of whether or not they are available on the terminal.

Groups of attributes (colors and fonts) cannot be split in the array. For instance, all foreground colors specified must be in adjacent locations in the array.

The first element of a group of attributes must be the default color or font of the terminal. For example, the first foreground color specified is usually **F_WHITE**, and the first background color specified is usually **B_BLACK**.

It is recommended that **sel_attr** only be called before **initscr**. If **sel_attr** is called after **initscr**, then the routine **setup_attr** should be called after calling **sel_attr**. If **sel_attr** is called after data has been added to a window, the values in the associated attribute array for that window may denote different attributes than the original attributes used when displaying the data (except **NORMAL** which remains constant). A subsequent refresh of the window shows the different attributes only if the data has been modified or if a total refresh has been forced by a previous call to **touchwin**.

The **sel_attr** routine always returns the value **OK**.

setterm (*name*)
char **name*;

The **setterm** routine sets the terminal characteristics to be those of the terminal specified by the *name* parameter. **setterm** is called by **initscr** so you do not normally have to use it unless you wish to use just the cursor motion optimizations.

setup_attr ()

The **setup_attr** routine creates the display attribute masks assigned to the attribute variables declared in the **cur01.h** header file. The priorities of the attributes determine how the masks are created.

This routine is called by **initscr** and is not normally called by applications. This routine should only be called following a call to **sel_attr** which follows a call to **initscr**.

standend ( )
wstandend (*win*)

The **standend** routine stops displaying characters in standout mode.

standout ( )
wstandout (*win*)

The **standout** routine starts displaying characters in standout mode. Any characters added to the window are put in standout mode on the terminal if the terminal has that capability. The first available attribute as determined by **sel_attr** is used for standout. This is normally the reverse attribute when the default display attribute priority is used.

The **standout** routine always returns the value **OK**.

WINDOW *subwin (*win*, *numlines*, *numcols*, *firstline*, *firstcol*)

The **subwin** routine creates a subwindow in the window pointed to by the *win* parameter. The subwindow has the number of lines specified by the *numlines* parameter and the number of columns specified by the *numcols* parameter. The new subwindow starts at the coordinates specified by the *firstline* and the *firstcol* parameters. Any change made to the window or the subwindow in the area covered by the subwindow is made to both windows.

The *firstline* and *firstcol* parameters are specified relative to the overall screen, not to the relative (0, 0) of the window specified by the *win* parameter.

If the *numlines* parameter is 0, then the lines dimension is set to (**LINES** - *firstline*). If the *numcols* parameter is 0, then the column dimension is set to (**COLS** - *firstcol*).

Upon successful completion, a pointer to the control structure for the new subwindow is returned.

The **subwin** routine returns **ERR** if the window specified by the *win* parameter already has a subwindow, or if there is insufficient storage for the new control structure.

superbox (*win*, *line*, *col*, *numlines*, *numcols*, *vert*, *hor*, *topl*, *topr*, *botl*, *botr*)
NLSCHAR *vert*, *hor*, *topl*, *topr*, *botl*,. *botr*;

The **superbox** routine draws a box on the window specified by the *win* parameter. The *line* and *col* parameters specify the starting coordinates for the box. The *numlines* parameter specifies the depth of the box. The *numcols* parameter specifies the width of the box. The *vert* parameter specifies the **NLSCHAR** to use for vertical delimiting. The *hor* parameter specifies the **NLSCHAR** to use for horizontal delimiting. The *topl*, *topr*, *botl*, and *botr* parameters specify the **NLSCHARs** to use for the top left corner, the top right corner, the bottom left corner, and the bottom right corner, respectively.

If the window specified by the *win* parameter is a **_SCROLLWIN** window and scrolling is not allowed, then the bottom right corner is not put on the window.

The **superbox** routine uses **addch** to place the characters on the window.

The **superbox** routine returns **ERR** if the defined box is outside the window, or an attempt is made to write to the lower right corner of the display when **scrollok** is off.

touchwin (*win*)

The **touchwin** routine makes it appear as if every location on the window specified by the *win* parameter has been changed. This is useful when overlapping windows are to be refreshed. A subsequent **refresh** request considers all portions of the window as potentially modified. If **touchwin** is not used, then only those positions of the window that have been addressed by an **addch** are inspected.

trackloc (*boolf*)

The **trackloc** routine turns on and off the tracking of the locator cursor on the screen. If the *boolf* parameter is **TRUE**, then locator tracking is turned on; if **FALSE**, then it is turned off. By default, locator tracking is initially turned on.

The keycode **KEY_LOCESC** is returned from **getch** when a locator report is input. The locator report is stored in the global **char** array **ESCSTR**, which is 128 bytes long.

Locator tracking is handled by the **ecpnin** routine.

tstp ( )

The **tstp** routine saves the current **tty** state and then put the process to sleep. When the process is restarted, the **tty** state is restored and then **wrefresh (curscr)** is called to redraw the screen. **initscr** sets the signal **SIGTSTP** to trap **tstp**.

The **tstp** routine always returns the value **OK**.

#include <cur04.h>

char *unctrl (*ch*)

The **unctrl** routine returns a string that represents the value of the *ch* parameter. Control characters become the lowercase equivalents preceded by a ^ (circumflex). Other letters are unchanged. This function supports only the P0 characters 0x00 through 0x7F.

Upon successful completion, a pointer to the string for the parameter character is returned.

vscroll (*win*, *numlines*, *numcols*)

The **vscroll** routine scrolls the viewport specified by the *win* parameter on the window.

The *numlines* parameter specifies the direction and amount to scroll up or down. If the *numlines* parameter is positive, the viewport scrolls down the number of lines specified. If the *numlines* parameter is negative, the viewport scrolls up the number of lines specified.

The *numcols* parameter specifies the direction and amount to scroll left or right. If the *numcols* parameter is positive, the viewport scrolls to the right the number of characters specified. If the *numcols* parameter is negative, then the viewport scrolls to the left the number of characters specified.

The **vscroll** routine always scrolls as much of a requested scroll as possible. Specifying a parameter with a magnitude larger than that of the underlying window is not an error.

The **vscroll** routine calls **touchwin** if any scrolling is done.

The **vscroll** routine returns **ERR** if the window specified by the *win* parameter is not a window created by a call to **newview**.

Files

/usr/lib/terminfo/?/* Compiled terminal capability data base.

Related Information

In this book: "curses" on page 3-51 and "terminfo" on page 4-148.

The discussion of Extended Curses in *AIX Operating System Programming Tools and Interfaces.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

fclose, fflush

Purpose

Closes or flushes a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int fclose (*stream***)**
FILE **stream*;

int fflush (*stream***)**
FILE **stream*;

Description

The **fclose** subroutine writes buffered data to the stream specified by the *stream* parameter and then closes the stream.

The **fclose** subroutine is automatically called for all open files when the **exit** system call is invoked.

The **fflush** subroutine writes any buffered data for the stream specified by the *stream* parameter and leaves the stream open.

Return Value

Upon successful completion, both the **fclose** and the **fflush** subroutines return a value of 0. If either of these subroutines fails for any reason, then it returns the value **EOF**.

Related Information

In this book: "close" on page 2-25, "exit, _exit" on page 2-40, "fopen, freopen, fdopen" on page 3-168, "setbuf, setvbuf" on page 3-330, and "standard i/o library" on page 3-342.

feof, ferror, clearerr, fileno

Purpose

Checks the status of a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int feof (*stream***)**
**FILE ****stream*;

int ferror (*stream***)**
**FILE ****stream*;

void clearerr (*stream***)**
**FILE ****stream*;

int fileno (*stream***)**
**FILE ****stream*;

Description

These macros inquire about the status of a stream.

The **feof** macro inquires about end-of-file. If **EOF** has previously been detected reading the input stream specified by the *stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **ferror** macro inquires about input/output errors. If an I/O error has previously occurred when reading from or writing to the stream specified by the *stream* parameter, a nonzero value is returned. Otherwise, a value of 0 is returned.

The **clearerr** macro resets the error indicator and the **EOF** indicator to 0 for the stream specified by the *stream* parameter.

The **fileno** macro returns the integer file descriptor associated with the input pointed to by the *stream* parameter.

Note: Since these routines are implemented as macros, they cannot be declared or redeclared.

Related Information

In this book: "open" on page 2-90, "fopen, freopen, fdopen" on page 3-168, and "standard i/o library" on page 3-342.

find_ipc_prof

Purpose

Finds a profile for an IPC queue.

Library

IPC Library (**libipc.a**)

Syntax

#include <drs.h>

int find_ipc_prof (*queue_name*, *l_key*, *r_key*, *nickname*)

char **queue_name*;
key_t **l_key*, **r_key*;
char **nickname*;

Description

The **find_ipc_prof** subroutine finds a profile for an IPC queue.

The *queue_name* parameter contains the name of an IPC queue. The *l_key* parameter points to the local key for an IPC queue. You must specify one or both of these values. The **find_ipc_prof** subroutine fails if both *queue_name* and *l_key* are **NULL**.

The *r_key* is a pointer from the local node to the IPC profile for a queue at a remote node. The *nickname* parameter points to the nickname or node ID, in hexadecimal, of the node where the IPC queue exists. A value of **NULL** indicates that the queue is on the local node.

The application does not supply values for the *r_key* and *nickname* parameters. The **find_ipc_prof** subroutine assigns values to these parameters when it returns. The application, however, must ensure that enough space is allocated to hold the return values.

Return Value

Upon successful completion, the function returns a 0, and *queue_name*, *l_key*, *r_key*, and *nickname* contain the values from the found profile. If an error occurs, **find_ipc_prof** returns a value from the following list:

DRS_ACCES	The required access permissions were denied.
DRS_BADLEN	An incorrect parameter was supplied.
DRS_NOREC	No record was found.
DRS_IO	An input/output error occurred.
DRS_AGAIN	Unable to start Profile Services.
DRS_BADF	An incorrect file descriptor was supplied.
DRS_BADK	An incorrect index key was supplied.
DRS_BDMSF	An incorrect file or table was supplied.
DRS_BOF	The beginning of the file was encountered.
DRS_DEADLK	A deadlock was detected.
DRS_EOF	The end of the file was encountered.
DRS_FAULT	An incorrect address was supplied.
DRS_FBIG	The maximum file size was exceeded.
DRS_IDRM	Identifier removed.
DRS_INBLCK	Big lock in Profile Services.
DRS_INTENT	Intentions denied.
DRS_ISDIR	A write to a directory was attempted.
DRS_MFILE	Too many files, tables, or indexes were open.
DRS_NFILE	The file table overflowed.
DRS_NOENT	No file or directory was found.
DRS_NOMEM	No memory is available.
DRS_NOSPC	No space is available on the device.
DRS_NOTDIR	Not a directory.
DRS_NOTIDX	Not an index.
DRS_PANIC	Abnormal termination occurred.
DRS_RCVRY	File needs recovery.

DRS_RECLEN	Record length is invalid.
DRS_ROFS	The file system to be accessed is read-only.

Related Information

In this book: "msgctl" on page 2-73, "create_ipc_prof" on page 3-40.2, and "del_ipc_prof" on page 3-64.1.

The **dsipc** command in *AIX Operating System Commands Reference.*

AIX Operating System Programming Tools and Interfaces.

floor, ceil, fmod, fabs

Purpose

Computes floor, ceiling, remainder, absolute value functions.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double floor (*x*)
double *x*;

double ceil (*x*)
double *x*;

double fmod (*x*, *y*)
double *x*, *y*;

double fabs (*x*)
double *x*;

Description

The **floor** subroutine returns the largest integer (as a **double**) not greater than the x parameter.

The **ceil** subroutine returns the smallest integer not less than the x parameter.

The **fmod** subroutine returns the remainder of $x \div y$. More precisely, this value is x if the y parameter is 0. Otherwise, it is the number f with the same sign as x such that $x = iy + f$ for some integer i, and $|f| < |y|$.

The **fabs** subroutine returns the absolute value of x, $|x|$.

Related Information

In this book: "abs" on page 3-6.

fopen, freopen, fdopen

Purpose

Opens a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

FILE *fopen (*path*, *type***)**
**char ****path*, ***type*;

FILE *freopen (*path*, *type*, *stream***)**
**char ****path*, ***type*;
FILE **stream*;

FILE *fdopen (*fildes*, *type***)**
int *fildes*;
**char ****type*;

Description

The **fopen** subroutine opens the file named by the *path* parameter and associates a stream with it. **fopen** returns a pointer to the **FILE** structure of this stream.

The *path* parameter points to a character string that contains the name of the file to be opened.

The *type* parameter points to a character string that has one of the following values:

`"r"`	Open the file for reading.
`"w"`	Truncate or create a new file for writing.
`"a"`	Append (open for writing at end of file, or create for writing).
`"r+"`	Open for update (reading and writing).
`"w+"`	Truncate or create for update.
`"a+"`	Append (open or create for update at end of file).

The **freopen** subroutine substitutes the named file in place of the open *stream*. The original *stream* is closed whether or not the **open** succeeds. **freopen** returns a pointer to the **FILE** structure associated with *stream*. The **freopen** subroutine is typically used to attach the pre-opened *streams* associated with **stdin**, **stdout**, and **stderr** to other files.

The **fdopen** subroutine associates a stream with a file descriptor obtained from an **open**, **dup**, **creat**, or **pipe** system call. These system calls open files but do not return pointers to **FILE** structures. Many of the standard I/O library subroutines require pointers to **FILE** structures. Note that the *type* of stream specified must agree with the mode of the open file.

When you open a file for update, you can perform both input and output operations on the resulting stream. However, an output operation cannot be directly followed by an input operation without an intervening **fseek** or **rewind**. Also, an input operation cannot be directly followed by an output operation without an intervening **fseek**, **rewind**, or an input operation that encounters the end of the file.

When you open a file for append (that is, when *type* is `"a"` or `"a+"`), it is impossible to overwrite information already in the file. You can use **fseek** to reposition the file pointer to any position in the file, but when output is written to the file, the current file pointer is ignored. All output is written at the end of the file and causes the file pointer to be repositioned to the end of the output.

If two separate processes open the same file for append, each process can write freely to the file without destroying the output being written by the other. The output from the two processes is intermixed in the order in which it is written to the file. Note that if the data is buffered, then it is not actually written until it is flushed.

If the **fopen** or **freopen** subroutine fails, a **NULL** pointer is returned.

Related Information

In this book: "creat" on page 2-27, "open" on page 2-90, "fclose, fflush" on page 3-163, "fseek, rewind, ftell" on page 3-196, "setbuf, setvbuf" on page 3-330, and "standard i/o library" on page 3-342.

fpfp

Purpose

Performs ANSI/IEEE floating-point operations.

Library

IEEE Support Library (**libieee.a**)

Syntax

#include <sys/FP.h>

Description

The **fpfp** subroutines perform floating-point operations that support ANSI/IEEE standard 754-1985 for binary floating-point arithmetic. This section assumes that you are familiar with the details of this standard.

The **fpfp** package supplies six abstract registers, each of which can hold a single-precision (**FP_FLOAT**) or double-precision (**FP_DOUBLE**) floating-point value. It also provides a status register that controls and reflects the result of the floating-point operations.

This interface is used by compilers to implement expressions that involve floating-point values. It is not intended to be used directly by user programs. By accessing the floating-point routines through a system-defined table, programs can transparently access either software floating-point routines, or a hardware floating-point processor, if one is installed in the system. The following five central-processor and floating-point hardware configurations are supported on the RT PC:

- 032 Microprocessor with no floating-point processor
- 032 Microprocessor with Floating-Point Accelerator
- 032 Microprocessor with Advanced Floating-Point Accelerator
- Advanced Processor Card (APC) with Advanced Floating-Point Accelerator
- Advanced Processor Card with a Motorola 68881 chip (APC/881).

Both Advanced Processor Card configurations take advantage of the floating-point processor's direct memory access capability (DMA).

On the RT PC, floating-point operations can be accessed in either of two modes:

Compatible Mode
Floating-point operations are invoked by calling subroutines in the kernel segment that either issue instructions to the floating-point processor, if one is present, or simulate the operations in software, if not. The RT PC C and FORTRAN compilers select compatible mode by default, so that programs compiled with the default options will run properly whether or not a floating-point processor is installed. This is the preferred mode, except for cases in which the greater performance of direct mode is essential.

Direct Mode
Instructions are issued directly to the hardware floating-point processor, yielding greater performance than compatible mode. Direct mode is available for the Floating-Point Accelerator and Advanced Floating-Point Accelerator. Direct mode is selected by specifying a flag to the compiler that you are using (see the specific language reference manual for details). A program that is compiled to use direct mode for one type of floating-point processor will not run properly on a machine that has a different floating-point processor or that has none at all.

Note: Although it is possible to access the hardware floating-point registers directly, doing so is strongly discouraged because this makes programs hardware-dependent. Programs that use compatible mode are independent not only of the characteristics of the floating-point hardware, but of whether a hardware floating-point processor is installed at all.

The remainder of this section gives details about compatible mode.

Table-Driven Interface

The primary interface to the compatible-mode subroutines is through a table that is located in the kernel segment. This table contains pointers to each of the compatible-mode subroutines. User processes have read-only access to this part of the kernel segment so that a simple subroutine call can be made, avoiding the overhead of performing a system call.

Although not recommended, these subroutines can be referenced directly by C programs. They must be invoked through the external array **_fpfpf**, which contains pointers to the subroutines. This array is indexed by values of the **FPFPI** enumeration data type.

For example, the C code to read the status register is:

```
FP_STATUS fpstat;
  . . .
*((unsigned *)&fpstat) = (*(unsigned (*) ( ))_fpfpf[(int)FP_getst]) ( );
```

Note that the subroutine **_FPgetst** is invoked by indexing the **_fpfpf** array with the **FP_getst** enumeration constant.

The following example adds double values in registers 4 and 5, stores the result in register 4, and then returns the result as well:

```
FP_DOUBLE drslt;
 . . .
*((FP_DOUBLE *)&drslt) = (*(FP_DOUBLE (*) ( ))_fpfpf[(int)FP_add])(4, 5);
```

The **FPFPI** enumeration data type is defined in the **sys/fpfpi.h** header file, and it contains the following values:

```
FP_rdf,      FP_rdd,      FP_i2f,      FP_i2d,
FP_cpf,      FP_cpfi,     FP_cpd,      FP_cpdi,
FP_f2d,      FP_f2di,     FP_d2f,      FP_d2fi,
FP_ngf,      FP_ngfi,     FP_ngd,      FP_ngdi,
FP_abf,      FP_abfi,     FP_abd,      FP_abdi,
FP_ntf,      FP_ntfi,     FP_ntd,      FP_ntdi,
FP_rnf,      FP_rnfi,     FP_rnd,      FP_rndi,
FP_trf,      FP_trfi,     FP_trd,      FP_trdi,
FP_flf,      FP_flfi,     FP_fld,      FP_fldi,
FP_cmf,      FP_cmfi,     FP_cmd,      FP_cmdi,
FP_adf,      FP_adfi,     FP_add,      FP_addi,
FP_sbf,      FP_sbfi,     FP_sbd,      FP_sbdi,
FP_mlf,      FP_mlfi,     FP_mld,      FP_mldi,
FP_dvf,      FP_dvfi,     FP_dvd,      FP_dvdi,
FP_rmf,      FP_rmfi,     FP_rmd,      FP_rmdi,
FP_sqf,      FP_sqfi,     FP_sqd,      FP_sqdi,
FP_csf,      FP_cfs,      FP_csd,      FP_cds,
FP_getst,    FP_setst,    FP_lmr,      FP_smr,
FP_tan,      FP_atan,     FP_2xm1,     FP_yl2x,
FP_ylp1,     FP_d2f2,     FP_f2d2,
FP_add3,     FP_adf3,     FP_sbd3,     FP_sbf3,
FP_mld3,     FP_mlf3,     FP_dvd3,     FP_dvf3,
FP_bsin,     FP_bsini,    FP_bcos,     FP_bcosi,
FP_btan,     FP_btani,    FP_bexp,     FP_bexpi,
FP_batan,    FP_batani,   FP_bacos,    FP_bacosi,
FP_basin,    FP_basini,   FP_blog,     FP_blogi,
FP_blog10,   FP_blog10i,  FP_blogb,    FP_blogbi,
FP_batan2,   FP_batan2i,  FP_bscalbi,
FP_bmodf,    FP_bmodfi,   FP_bmodd,    FP_bmoddi,
FP_lmrd,     FP_smrd,
FP_rdfa,   FP_rdda,   FP_cpfa,    FP_cpda,
FP_f2da,   FP_d2fa,   FP_cmfa,    FP_cmda,
FP_adfa,   FP_adda,   FP_sbfa,    FP_sbda,
FP_mlfa,   FP_mlda,   FP_dvfa,    FP_dvda,
FP_sqfa,   FP_sqda,   FP_bsina,   FP_bcosa,
FP_type
```

The operations named **FP_***xxxx* in the preceding table correspond to the subroutines named **_FP***xxxx* in "Subroutines" on page 3-176.

Note that the **sys/FP.h** header file **#include**s all of the other header files required by these floating-point routines. Therefore, your programs need to **#include** only this one file.

Fixed Entry Points

In order to minimize the overhead of calling the floating-point routines that are used most frequently, an alternate interface that uses a nonstandard calling sequence is available for some of the operations. Because this is a nonstandard interface, you should not attempt to use it in programs written in high-level languages. Use this interface in assembler language programs instead.

These entry points are located at fixed locations in low memory. This fact allows programs to branch directly to the routines with a **balax** instruction instead of using the table-driven interface, avoiding four 032 Microprocessor or APC (032/APC) instructions that set up the constant pool pointer and get the address of the routine indirectly.

The subroutines are the same as those described under "Subroutines" on page 3-176, except that the first parameter is passed in general-purpose register 2 of the 032/APC, and the second parameter is passed in register 3 (for a register number, a float immediate value, or the address of an immediate value) or in registers 3 and 4 (for a double immediate value). The third parameter is passed in register 4.

The following table lists the address of each fixed entry point and the 032/APC general-purpose registers that are modified. The Multiplier Quotient (MQ) register may also be modified. Other registers need not be saved before calling the routine.

Operation	Entry Point	Registers Modified
_FPadd	0x1300	2 3
_FPadda	0x1A00	2 3 4 5
_FPaddi	0x1340	2 3 4 5
_FPadf	0x1280	2 3
_FPadfa	0x19C0	2 3
_FPadfi	0x12C0	2 3
_FPbcos	0x1CC0	2 3 4 5
_FPbcosa	0x1E80	0 2 3 4 5
_FPbcosi	0x1D00	2 3 4 5
_FPbsin	0x1D40	2 3 4 5
_FPbsina	0x1E40	0 2 3 4 5
_FPbsini	0x1D80	2 3 4 5
_FPcmd	0x1740	0 2 3 4 5
_FPcmda	0x1980	0 2 3 4 5
_FPcmdi	0x1780	0 2 3 4 5
_FPcmf	0x16C0	0 2 3 4 5
_FPcmfa	0x1940	0 2 3 4 5
_FPcmfi	0x1700	0 2 3 4 5
_FPcpd	0x1100	2 3
_FPcpda	0x1880	2 3 4
_FPcpdi	0x1140	2 3 4
_FPcpf	0x1080	2 3
_FPcpfa	0x1840	2 3
_FPcpfi	0x10C0	2 3
_FPd2f	0x1200	0 2 3
_FPd2fa	0x1900	0 2 3
_FPd2fi	0x1240	0 2 3 4
_FPdvd	0x1600	2 3
_FPdvda	0x1B80	2 3 4 5
_FPdvdi	0x1640	2 3 4 5
_FPdvf	0x1580	2 3
_FPdvfa	0x1B40	2 3
_FPdvfi	0x15C0	2 3
_FPf2d	0x1180	0 2 3
_FPf2da	0x18C0	0 2 3
_FPf2di	0x11C0	0 2 3
_FPgetst	0x1680	2 3
_FPmld	0x1500	2 3
_FPmlda	0x1B00	2 3 4 5
_FPmldi	0x1540	2 3 4 5
_FPmlf	0x1480	2 3
_FPmlfa	0x1AC0	2 3
_FPmlfi	0x14C0	2 3
_FPrdd	0x1040	2 3
_FPrdda	0x1800	2 3 4
_FPrdf	0x1000	2 3
_FPrdfa	0x17C0	2 3
_FPsbd	0x1400	2 3
_FPsbda	0x1A80	2 3 4 5
_FPsbdi	0x1440	2 3 4 5
_FPsbf	0x1380	2 3
_FPsbfa	0x1A40	2 3
_FPsbfi	0x13C0	2 3
_FPsqd	0x1C40	0 2 3
_FPsqda	0x1E00	2 3 4 5
_FPsqdi	0x1C80	2 3 4 5
_FPsqf	0x1BC0	0 2 3
_FPsqfa	0x1DC0	0 2 3
_FPsqfi	0x1C00	0 2 3 4 5

Example

The following assembler subroutine adds 1.0 and 2.0 in single precision and returns the result. The subroutine can be called from a C-language program. See "Parameters" on page 3-175 about specifying whether or not a floating-point operation returns the result.

```
        .set   NORESULT,0x08

        .text
        .globl .add12
.add12:
        cal    1,-44(1)
        stm    14,0(1)       # Save 032 registers 14 and 15
        lr     14,0

# Load 2.0 into float register 0; don't return the result
        cau    3,0x4000(0)   # Specify 2.0 as second parameter
        balax  0x10C0        # FP_cpfi - copy float immediate
          lis  2,0|NORESULT  # Specify float reg 0 as first parameter;
                             #    don't return the result

# Add 1.0 to float register 0; return the result
        cau    3,0x3F80(0)   # Specify 1.0 as second parameter
        balax  0x12C0        # FP_adfi - add float immediate
          lis  2,0           # Specify float reg 0 as first parameter;
                             #    DO return the result

# The result is now in 032 register 2
# Return the result in register 2  (C-language calling convention)
        lm     14,0(1)
        brx    15
          cal  1,44(1)

        .data  3
        .globl _add12
_add12  .long  .add12
```

Parameters

The following parameter declarations apply to all of the floating-point subroutines:

int	*r1*, *r2*, *r3*;
int	*ival*, *mask*;
FP_FLOAT	*fval*, **fptr*, **frsltp*;
FP_DOUBLE	*dval*, **dptr*, **drsltp*, **dbuf*;
unsigned char	**dec*;
FP_STATUS	*status*;

The *r1*, *r2*, and *r3* parameters are integers in the range 0 to 5, denoting one of the six abstract floating-point registers.

In addition, the *r1* parameter can be logically OR-ed with the value **NORETBIT**, called the ***no-result flag***, which permits greater performance by suppressing the return value from the subroutine when it is not needed immediately. (**NORETBIT** is defined in the **sys/FP.h** header file.) This allows the central processor and a floating-point processor to operate concurrently: after issuing a request to the floating-point processor, the central processor normally waits for it to finish the operation and return the result. If the no-result flag is set, then the central processor does not wait, but continues running while the floating-point processor performs the requested operation. The value returned by an operation is undefined if the no-result flag is set.

Most of the operations store the result into the floating-point register specified by the *r1* parameter. However, operations whose names end with the letter **a** are passed pointers to an immediate value (*fptr* or *dptr*) and to the memory location where the result is to be stored (*frsltp* or *drsltp*). If the no-result flag is set for one of these operations, then the result pointer does not need to be specified.

Warning: If the no-result flag is not set and the *frsltp* or *drsltp* parameter is not specified or does not point to a valid float or double variable, then the results of the operation are unpredictable. For example, if the parameter points to a location outside of the process's allocated address space, then the process receives a memory fault.

Subroutines

The following subroutines are grouped by function. The subroutine names shown are those that appear in the Floating-Point Library (**libfp.a**). See "Table-Driven Interface" on page 3-171 for the recommended interface to these routines. The no-result flag applies to all of them, except as noted.

Load and Store Operations

FP_FLOAT	**_FPrdf** (*r1*)
void	**_FPrdfa** (*r1*, *frsltp*)
FP_DOUBLE	**_FPrdd** (*r1*)
void	**_FPrdda** (*r1*, *drsltp*)

Reads the float or double value that is in register *r1*, and either returns the result or stores it into the memory location pointed to by the *rsltp* parameter.

FP_FLOAT	**_FPi2f** (*r1*, *ival*)
FP_DOUBLE	**_FPi2d** (*r1*, *ival*)

Converts the integer value to float or double, stores the converted value into register *r1*, and returns that value.

FP_FLOAT	**_FPcpf (*r1*, *r2*)**
FP_FLOAT	**_FPcpfi (*r1*, *fval*)**
void	**_FPcpfa (*r1*, *fptr*, [*frsltp*])**
FP_DOUBLE	**_FPcpd (*r1*, *r2*)**
FP_DOUBLE	**_FPcpdi (*r1*, *dval*)**
void	**_FPcpda (*r1*, *dptr*, [*drsltp*])**

Copies into register *r1* the float or double value from: register *r2*, the *val* parameter, or the memory location pointed to by *ptr*. The result is either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_DOUBLE	**_FPf2d (*r1*)**
FP_DOUBLE	**_FPf2di (*r1*, *fval*)**
void	**_FPf2da (*r1*, *fptr*, [*drsltp*])**

Converts to double the float value in: register *r1*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_FLOAT	**_FPd2f (*r1*)**
FP_FLOAT	**_FPd2fi (*r1*, *dval*)**
void	**_FPd2fa (*r1*, *dptr*, [*frsltp*])**

Converts to float the double value in: register *r1*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_DOUBLE	**_FPf2d2 (*r1*, *r2*)**
FP_FLOAT	**_FPd2f2 (*r1*, *r2*)**

Converts the float or double value in register *r2* to double or float, stores it into register *r1*, and returns that value. The value in register *r2* remains unmodified.

Multiple-Register Load and Store Operations

_FPlmr Loads multiple floating-point registers with values stored in memory.

_FPsmr Stores multiple floating-point registers into a buffer in memory.

These subroutines require a nonstandard calling sequence that can be performed in assembler language programs, but not using high-level languages.

Both subroutines are passed the following parameters in 032/APC general-purpose registers:

Reg	**Contents**
12	Specifies the address of the save area.

11 The low-order 16 bits contain a bit mask that specifies the floating-point registers to load or store. Single-precision **fpfp** registers 0 through 5 are selected by setting the even bits 0, 2, 4, 6, 8, and 10, where bit 0 is the least significant bit of the mask. The double-precision **fpfp** registers 0 through 5 are selected by pairs of bits: 0-1, 2-3, 4-5, 6-7, 8-9, and 10-11. For example, setting bits 4 and 5 selects double-precision **fpfp** register 2. This register mapping is shown in Figure 3-2 on page 3-190.2.

Both operations modify 032/APC registers 0, 11, 12, and 13.

void _FPlmrd (*mask*, *dbuf*)
void _FPsmrd (*mask*, *dbuf*)

Loads or stores multiple *double-precision* floating-point registers. The *dbuf* parameter points to the beginning of the register save area. The *mask* parameter is a bit mask that specifies the registers to load or store. Registers 0 to 31 are selected by bits 0 to 31, respectively, where bit 0 is the least significant bit of the mask.

Unary Operations

FP_FLOAT _FPngf (*r1*)
FP_FLOAT _FPngfi (*r1*, *fval*)
FP_DOUBLE _FPngd (*r1*)
FP_DOUBLE _FPngdi (*r1*, *dval*)

Negates the float or double value in register *r1* or in the *val* parameter, stores it into register *r1*, and returns that value.

FP_FLOAT _FPabf (*r1*)
FP_FLOAT _FPabfi (*r1*, *fval*)
FP_DOUBLE _FPabd (*r1*)
FP_DOUBLE _FPabdi (*r1*, *dval*)

Makes the float or double value in register *r1* or in the *val* parameter positive without changing its magnitude, stores it into register *r1*, and returns that value.

FP_FLOAT _FPntf (*r1*)
FP_FLOAT _FPntfi (*r1*, *fval*)
FP_DOUBLE _FPntd (*r1*)
FP_DOUBLE _FPntdi (*r1*, *dval*)

Rounds the float or double value in register *r1* or in the *val* parameter to an integer in floating-point format, stores the result into register *r1*, and returns that value. The rounding performed depends on the rounding mode specified in the floating-point status word. (See "Status Register Operations" on page 3-186.)

int _FPrnf (*r1*)
int _FPrnfi (*r1*, *fval*)
int _FPrnd (*r1*)
int _FPrndi (*r1*, *dval*)

Copies the *val* parameter, if given, to register *r1*, then returns the integer value that is nearest to the float or double value in register *r1*. Note that the value in register *r1* is not converted to an integer. The rounding performed is independent of the rounding mode specified in the floating-point status register. The no-result flag does not apply to this operation. (See "Parameters" on page 3-175 for an explanation of the no-result flag.)

int _FPtrf (*r1*)
int _FPtrfi (*r1*, *fval*)
int _FPtrd (*r1*)
int _FPtrdi (*r1*, *dval*)

Copies the *val* parameter, if given, to register *r1*, then returns the integer part of the float or double value in register *r1*. Note that the value in register *r1* is not converted to an integer. The rounding performed is independent of the rounding mode specified in the floating-point status register. The no-result flag does not apply to this operation. (See "Parameters" on page 3-175 for an explanation of the no-result flag.)

int _FPflf (*r1*)
int _FPflfi (*r1*, *fval*)
int _FPfld (*r1*)
int _FPfldi (*r1*, *dval*)

Copies the *val* parameter, if given, to register *r1*, then returns the largest integer less than or equal to the float or double value in register *r1*. Note that the value in register *r1* is not converted to an integer. The rounding performed is independent of the rounding mode specified in the floating-point status register. The no-result flag does not apply to this operation. (See "Parameters" on page 3-175 for an explanation of the no-result flag.)

Comparison Operations

int _FPcmf (*r1*, *r2*)
int _FPcmfi (*r1*, *fval*)
int _FPcmfa (*r1*, *fptr*)
int _FPcmd (*r1*, *r2*)
int _FPcmdi (*r1*, *dval*)
int _FPcmda (*r1*, *dptr*)

Compares the float or double value in register *r1* to the float or double value in: register *r2*, the *val* parameter, or the memory location pointed to by *ptr*. The value returned is **LESSTHAN**, **EQUAL**, **GREATER** or **MININT**, depending on whether the value in register *r1* is less than, equal to, greater than, or unordered with the other value, respectively. In addition, the 032/APC test bit is set to 1 if the comparison is unordered, and it is set to 0 otherwise.

If the *r1* parameter is logically OR-ed with the value **ExceptOnUnordered**, and if the operands are unordered, then an invalid operation exception occurs. (See "Status Register Operations" on page 3-186 for details about floating-point exceptions.)

The no-result flag does not apply to these comparison operations. (See "Parameters" on page 3-175 for an explanation of the no-result flag.) Also, **_FPcmfa** and **_FPcmda** *return* the result, rather than storing it into a memory location.

Simple Arithmetic Operations

FP_FLOAT	**_FPadf** (*r1*, *r2*)
FP_FLOAT	**_FPadfi** (*r1*, *fval*)
void	**_FPadfa** (*r1*, *fptr*, [*frsltp*])
FP_DOUBLE	**_FPadd** (*r1*, *r2*)
FP_DOUBLE	**_FPaddi** (*r1*, *dval*)
void	**_FPadda** (*r1*, *dptr*, [*drsltp*])

Adds to the float or double value in register *r1* the float or double value in: register *r2*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_FLOAT	**_FPsbf** (*r1*, *r2*)
FP_FLOAT	**_FPsbfi** (*r1*, *fval*)
void	**_FPsbfa** (*r1*, *fptr*, [*frsltp*])
FP_DOUBLE	**_FPsbd** (*r1*, *r2*)
FP_DOUBLE	**_FPsbdi** (*r1*, *dval*)
void	**_FPsbda** (*r1*, *dptr*, [*drsltp*])

Calculates the difference between the float or double value in register *r1* and the float or double value in: register *r2*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_FLOAT	**_FPmlf** (*r1*, *r2*)
FP_FLOAT	**_FPmlfi** (*r1*, *fval*)
void	**_FPmlfa** (*r1*, *fptr*, [*frsltp*])
FP_DOUBLE	**_FPmld** (*r1*, *r2*)
FP_DOUBLE	**_FPmldi** (*r1*, *dval*)
void	**_FPmlda** (*r1*, *dptr*, [*drsltp*])

Multiplies the float or double value in register *r1* by the float or double value in: register *r2*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_FLOAT	**_FPdvf** (*r1*, *r2*)
FP_FLOAT	**_FPdvfi** (*r1*, *fval*)
void	**_FPdvfa** (*r1*, *fptr*, [*frsltp*])
FP_DOUBLE	**_FPdvd** (*r1*, *r2*)
FP_DOUBLE	**_FPdvdi** (*r1*, *dval*)
void	**_FPdvda** (*r1*, *dptr*, [*drsltp*])

Divides the float or double value in register *r1* by the float or double value in: register *r2*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_FLOAT	**_FPrmf** (*r1*, *r2*)
FP_FLOAT	**_FPrmfi** (*r1*, *fval*)
FP_DOUBLE	**_FPrmd** (*r1*, *r2*)
FP_DOUBLE	**_FPrmdi** (*r1*, *dval*)

Calculates the IEEE remainder that results when the float or double value in register *r1* is divided by the float or double value in register *r2* or in the *val* parameter, stores the result into register *r1*, and then returns the result. The result is rounded to the nearest value. See also "_fpbmodf, _fpbmodfi, _fpbmodd, _fpbmoddi" on page 3-184.

FP_FLOAT   _FPsqf (*r1*)
FP_FLOAT   _FPsqfi (*r1*, *fval*)
void   _FPsqfa (*r1*, *fptr*, [*frsltp*])
FP_DOUBLE   _FPsqd (*r1*)
FP_DOUBLE   _FPsqdi (*r1*, *dval*)
void   _FPsqda (*r1*, *dptr*, [*drsltp*])

Calculates the square root of the float or double value in: register *r1*, the *val* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

Three-Operand Arithmetic Operations

FP_FLOAT   _FPadf3 (*r1*, *r2*, *r3*)
FP_DOUBLE   _FPadd3 (*r1*, *r2*, *r3*)

Adds the float or double values in registers *r2* and *r3*, stores the result into register *r1*, and then returns the result.

FP_FLOAT   _FPsbf3 (*r1*, *r2*, *r3*)
FP_DOUBLE   _FPsbd3 (*r1*, *r2*, *r3*)

Subtracts the float or double value in register *r3* from the value in register *r2*, stores the result into register *r1*, and then returns the result.

FP_FLOAT   _FPmlf3 (*r1*, *r2*, *r3*)
FP_DOUBLE   _FPmld3 (*r1*, *r2*, *r3*)

Multiplies the float or double values in registers *r2* and *r3*, stores the result into register *r1*, and then returns the result.

FP_FLOAT   _FPdvf3 (*r1*, *r2*, *r3*)
FP_DOUBLE   _FPdvd3 (*r1*, *r2*, *r3*)

Divides the float or double value in register *r2* by the value in register *r3*, stores the result into register *r1*, and then returns the result.

Packed Decimal Conversion Operations

FP_FLOAT   _FPcsf (*r1*, *dec*)
FP_DOUBLE   _FPcsd (*r1*, *dec*)

Converts the packed decimal string pointed to by the *dec* parameter to float or double, stores the result into register *r1*, and then returns the result. The *dec* parameter points to a 10-byte packed decimal number, the first byte of which is 0 (for a positive value) or 0x80 (for a negative value). The following 9 bytes containing 2 decimal digits each; the left half-byte of each byte contains the more significant digit.

void _FPcfs (*r1*, *dec*)
void _FPcds (*r1*, *dec*)

Converts the float or double value in register *r1* to packed decimal format as described for **_FPcsf** and **_FPcsd**. In the special cases of + ∞, -∞, and **NaN**, the first byte is set to 0x7F, 0xFF, or 0x0F, respectively.

Miscellaneous Operation

int _FPtype ( )

Returns one of the following values to identify the current floating-point hardware configuration:

0 Emulation (no floating-point processor)
1 032 Microprocessor with Floating-Point Accelerator
2 Advanced Processor Card with 68881
4 032 Microprocessor with Advanced Floating-Point Accelerator
12 Advanced Processor Card with Advanced Floating-Point Accelerator.

Transcendental Operations

These operations implement a version of the transcendental routines developed at the University of California at Berkeley, which handle parameters and generate results in a manner consistent with the IEEE floating-point standard.

Note: These operations are only available in AIX version 2.1 and later. Programs that use them will not run on earlier versions of AIX.

FP_DOUBLE _FPbsin (*r1*, *r2*)
FP_DOUBLE _FPbsini (*r1*, *dval*)
void _FPbsina (*r1*, *dptr*, [*drsltp*])
FP_DOUBLE _FPbcos (*r1*, *r2*)
FP_DOUBLE _FPbcosi (*r1*, *dval*)
void _FPbcosa (*r1*, *dptr*, [*drsltp*])
FP_DOUBLE _FPbtan (*r1*, *r2*)
FP_DOUBLE _FPbtani (*r1*, *dval*)

Calculates the sine, cosine, or tangent of the double value in: register *r2*, the *dval* parameter, or the memory location pointed to by *ptr*. The result is stored into register *r1* and is then either returned or stored into the memory location pointed to by the *rsltp* parameter.

FP_DOUBLE _FPbasin (*r1*, *r2*)
FP_DOUBLE _FPbasini (*r1*, *dval*)
FP_DOUBLE _FPbacos (*r1*, *r2*)
FP_DOUBLE _FPbacosi (*r1*, *dval*)
FP_DOUBLE _FPbatan (*r1*, *r2*)
FP_DOUBLE _FPbatani (*r1*, *dval*)
FP_DOUBLE _FPbatan2 (*r1*, *r2*)
FP_DOUBLE _FPbatan2i (*r1*, *dval*)

The first six subroutines calculate the arcsine, arccosine, and arctangent of the double value in register *r2* or in the *dval* parameter, store the result into register *r1*, and then return the result. **_FPbatan2** and **_FPbatan2i** calculate the arctangent of $y \div x$, where *y* (the value in register *r1*) and *x* (the value in register *r2* or in the *dval* parameter) are both double values. The result is then stored into register *r1* and returned.

FP_DOUBLE _FPblog (*r1*, *r2*)
FP_DOUBLE _FPblogi (*r1*, *dval*)
FP_DOUBLE _FPblog10 (*r1*, *r2*)
FP_DOUBLE _FPblog10i (*r1*, *dval*)

Calculates the natural or base-10 logarithm of the double value in register *r2* or in the *dval* parameter, stores the result into register *r1*, and then returns the result.

FP_DOUBLE _FPbexp (*r1*, *r2*)
FP_DOUBLE _FPbexpi (*r1*, *dval*)

Calculates the value of e^x, where register *r2* or the *dval* parameter contains a double value representing the exponent (*x*). The result is stored into register *r1* and returned.

FP_DOUBLE _FPbmodf (*r1*, *r2*)
FP_DOUBLE _FPbmodfi (*r1*, *fval*)
FP_DOUBLE _FPbmodd (*r1*, *r2*)
FP_DOUBLE _FPbmoddi (*r1*, *dval*)

Calculates the modulo remainder that results when the float or double value in register *r1* is divided by the float or double value in register *r2* or in the *val* parameter, stores the result into register *r1*, and then returns the result. The result is rounded toward zero. See also "_fprmf, _fprmfi, _fprmd, _fprmdi" on page 3-181.

FP_DOUBLE _FPbscalbi (*r1*, *ival*)

Calculates the value of $y \cdot 2^N$ for integral values of N without computing 2^N. Register *r1* contains *y*, and the *ival* specifies *N*. The result of this operation is stored into register *r1* and is returned as the value of the subroutine.

FP_DOUBLE _FPblogb (*r1*, *r2*)
FP_DOUBLE _FPblogbi (*r1*, *dval*)

Extracts the unbiased exponent from the double value in register *r2* or in the *dval* parameter, stores the result into register *r1*, and then returns the result. The result is a signed integer in double-precision floating-point format, except that the **logb** of a NaN is a Nan, the **logb** of ∞ is $+\infty$, and the **logb** of 0 is $-\infty$. Taking the **logb** of 0 also causes a division by zero exception to occur. (See "Status Register Operations" on page 3-186 for details about floating-point exceptions.) If *x* is positive and finite, then the expression **_FPbscalbi (*r*, -_FPblogbi (*s*, *x*))** lies strictly between 0 and 2; it is less than 1 only when *x* is denormalized.

Additional Transcendental Operations

These operations are the core calculations for other logarithmic, hyperbolic, and trigonometric functions, and they can be used as partial steps in calculating the result desired by the user. A more convenient set of transcendentals is discussed under "Transcendental Operations" on page 3-183. The basic transcendentals are not implemented by any of the floating-point processors, but by software routines only.

FP_DOUBLE _FPtan (*r1*, *r2*)

Calculates the partial tangent of an angle measured in radians. Register *r1* contains the angle, which must be a double value in the range $0 < r1 \leq \pi \div 4$. If the value in register *r1* is outside this range, then an invalid operation exception occurs, and no operation is performed. The result of this operation consists of 2 double values. The sine of the angle is stored into register *r1* and is returned as the value of the subroutine. The cosine is stored into register *r2*. To calculate the tangent, divide the sine by the cosine.

FP_DOUBLE _FPatan (*r1*, *r2*)

Calculates the arctangent of $y \div x$. This operation requires a double value in both registers *r1* and *r2*. These arguments must be ordered such that the value in *r1* (*x*) is greater than the value in *r2* (*y*), which is greater than 0. If the values are not ordered in this manner, then an invalid operation exception occurs, and no operation is performed. The result is stored into register *r1* and is returned as the value of the subroutine.

FP_DOUBLE _FP2xm1 (*r1*)

Calculates the value of 2^x - 1. Register *r1* contains the exponent (*x*), which must a double value be in the range $0 \leq r1 \leq 0.5$. If the value in register *r1* is not in this range, then an invalid operation exception occurs, and no operation is performed. The result of this operation is stored into register *r1* and is returned as the value of the subroutine.

FP_DOUBLE _FPyl2x (*r1*, *r2*)

Calculates the value of $y \cdot \log_2 x$. This operation requires a double value in both registers *r1* and *r2*. Register *r1* contains the exponent (*x*), which must be a double value the range $0 < r1 < +\infty$. If the value in register *r1* is outside this range, then an invalid operation exception occurs, and no operation is performed. Register *r2* contains the multiplier (*y*), which can be any valid double value. The result of this operation is stored into register *r1* and is returned as the value of the subroutine.

FP_DOUBLE _FPylp1 (*r1*, *r2*)

Calculates the value of $y \cdot \mathbf{log}_2\ (x+1)$. This operation requires a double value in both registers *r1* and *r2*. Register *r1* contains *x*, which must be a double value in the range:

$$0 < r1 < \left(1 - \frac{\sqrt{2}}{2}\right)$$

If the value in register *r1* is outside this range, then an invalid operation exception occurs, and no operation is performed. Register *r2* contains the multiplier (*y*), which can be any valid double value. The result of the operation is stored into register *r1* and is returned as the value of the subroutine. This subroutine provides more accurate results than **_FPyl2x** for *x* values close to 1.

Status Register Operations

The floating-point status is kept in a **FP_STATUS** structure, which is defined with a **typedef** statement in the **sys/fpfp.h** header file, and which contains the following fields:

`unsigned kill`	: 1	Signal **SIGFPE** on an exception
`unsigned xcp_flag`	: 1	An exception occurred
`unsigned io_flag`	: 1	An invalid operation occurred
`unsigned io_xpt`	: 1	Enable signal on invalid operation
`unsigned dz_flag`	: 1	A division by 0 occurred
`unsigned dz_xpt`	: 1	Enable signal on division by 0
`unsigned of_flag`	: 1	An overflow occurred

```
unsigned of_xpt      : 1  Enable signal on overflow
unsigned uf_flag     : 1  An underflow occurred
unsigned uf_xpt      : 1  Enable signal on underflow
unsigned precision   : 1  Rounding precision
unsigned cmp_rslt    : 2  Comparison result
unsigned rnd_mode    : 2  Rounding mode
unsigned ir_flag     : 1  An inexact result occurred
unsigned ir_xpt      : 1  Enable signal on inexact result
```

When set, the **kill** field enables exception trapping. The **io_xpt**, **dz_xpt**, **of_xpt**, **ir_xpt**, and **uf_xpt** fields enable trapping for each of the five exception types. Thus, for an exception to be trapped, both **kill** and the *xx*_**xpt** field for that exception must be set. See "Exception Handling" on page 3-188 for detailed information about exceptions and traps.

When an exception occurs for which trapping is not enabled, the fields **io_flag**, **dz_flag**, **of_flag**, **ir_flag**, and **uf_flag** are set to indicate the exception or exceptions that occurred. The **xcp_flag** field is also set to indicate that an exception occurred. The **xcp_flag** and *xx*_**flag** fields are not automatically reset to 0; this must be done explicitly by calling **_FPsetst**.

The **precision** field selects the rounding precision to be used. If it is set to 0, then all results are rounded to the greatest precision available with the floating-point hardware being used. (This is extended precision for the 68881 and double for all others.) If **precision** is set to 1, then all results are rounded to double precision. If the 68881 is being used, then setting any *xx*_**xpt** field for trapping exceptions also causes **precision** to be set to 1. This ensures that exception handlers always receive the same results regardless of the hardware configuration. Note that rounding to double precision slows down floating-point operations on the 68881.

The **cmp_rslt** field of the status word specifies the result of a comparison operation, and it is set to one of the following values:

0	Less than
1	Equal
2	Greater than
3	Unordered.

The **rnd_mode** field of the status word specifies the rounding mode, and it can be set to one of the following values:

FP_NEAR	Rounds to the nearest value. This is the default.
FP_ZERO	Rounds toward 0.
FP_UP	Rounds toward +∞.
FP_DOWN	Rounds toward -∞.

FP_STATUS _FPgetst ( )

Returns the floating-point status word reflecting the results of the most recent floating-point operation.

void _FPsetst (*status*)

Sets the floating-point status word to the value provided by the *status* parameter.

Exception Handling

When an error is detected during a floating-point operation, an ***exception*** occurs. Each exception is one of five types, as specified by the IEEE standard: invalid operation, overflow, underflow, division by zero, and inexact result.

The standard also defines a ***default result*** that is returned as the value of the operation that caused the exception. Alternatively, you can specify a special action to be taken when an exception occurs. Performing a user-defined action when an exception occurs is called ***exception trapping***, and a user-defined trap routine is called a ***trap handler***. However, the default result is sufficient for most applications, and specifying a trap handler is usually not necessary.

When an exception occurs, the floating-point operation checks the trap-enabling fields of the floating-point status word. (See "Status Register Operations" on page 3-186 for details about the trap-enabling fields.) If either the **kill** field or the *xx*_**xpt** field for the exception type is not set, then the exception is not trapped and the following default actions take place:

- The **xcp_flag** field is set to indicate that an exception occurred.
- The appropriate *xx*_**flag** field is set to indicate the exception type.
- The default result specified by the IEEE standard is returned to the application program.

If both the **kill** field and the *xx*_**xpt** field are set, then trapping is enabled for the given exception type. In this case, the **SIGFPE** signal is sent to the process. (AIX signals are discussed in "signal" on page 2-145.) You can write your own signal handler, which must determine the exception that occurred and take appropriate action, or you can use the facility provided by the **ieeetrap** subroutine.

Note: If the status word setting enables the trapping of one or more exception types and no signal handler is specified for **SIGFPE** (either a user-supplied handler or the one installed by **ieeetrap**), then the default action **(SIG_DFL)** is taken for the signal, which terminates the process.

The **ieeetrap** subroutine, which is located in the IEEE Support Library (**libieee.a**), allows you to specify a trap handler for each of the five exception types. **ieeetrap** functions like the **signal** system call and has the following syntax:

> **int (*ieeetrap (***exceptiontype*, *action***)) ( )**
> **int** *exceptiontype***;**
> **int (****action***) ( );**

The *exceptiontype* parameter identifies the type of exception and is one of the following values:

FP_INV_OPER	Invalid operation
FP_OVERFLOW	Overflow
FP_UNDERFLOW	Underflow
FP_DIVIDE	Division by zero
FP_INEXACT	Inexact result.

The *action* parameter is one of the following:

SIG_IGN Sets the status word to disable trapping of the specified exception type. This is the default action for a given exception type if an action is not specified for it.

SIG_DFL Causes the process to be terminated upon an exception of the specified type. The process is terminated by setting the action for the **SIGFPE** signal to **SIG_DFL** and sending **SIGFPE** to the process. This allows the parent process to determine that the process terminated due to a floating-point exception.

handler Causes the subroutine with the address given by *handler* to be called when an exception of the specified type occurs.

Warning: Unpredictable results can occur if you use **ieeetrap** to disable trapping of a given exception type (**SIG_IGN**) and then enable trapping by using **_FPsetst** to set the status word. Instead, use **ieeetrap** to set a trap handler for the exception type.

The **ieeetrap** subroutine establishes a signal handler for **SIGFPE** that selects one of five actions based on the exception type. It also sets *action* as the action to be taken when an *exceptiontype* exception occurs.

Upon successful completion, the **ieeetrap** subroutine returns the previous value of *action* for the specified *exceptiontype.* If an error occurs, then the value **BADSIG** is returned.

If the *action* parameter to **ieeetrap** specifies a *handler* subroutine, then an exception of the specified type causes the *handler* subroutine to be called with the following syntax:

#include <sys/FP.h>
#include <signal.h>

handler **(***sig***,** *code***,** *scp***)**
int *sig***,** *code***;**
struct sigcontext ****scp*;**

The parameters are the same as those passed to the signal handler. (See "sigvec" on page 2-156 for a detailed description.) Note that the value of the *sig* parameter is always **SIG_FPE**.

The following discussion traces the hierarchy of structures pointed to by the *scp* parameter. Use Figure 3-1 on page 3-190.1 as a guide for this discussion.

The **sigcontext** structure, which is defined in the **sys/signal.h** header file, contains a member named **fpvmp**, which points to a **fpvmach** structure. The **fpvmach** structure describes the state of the virtual floating-point machine at the time of the exception. This structure is defined in the **ieeetrap.h** header file, and it contains the following members:

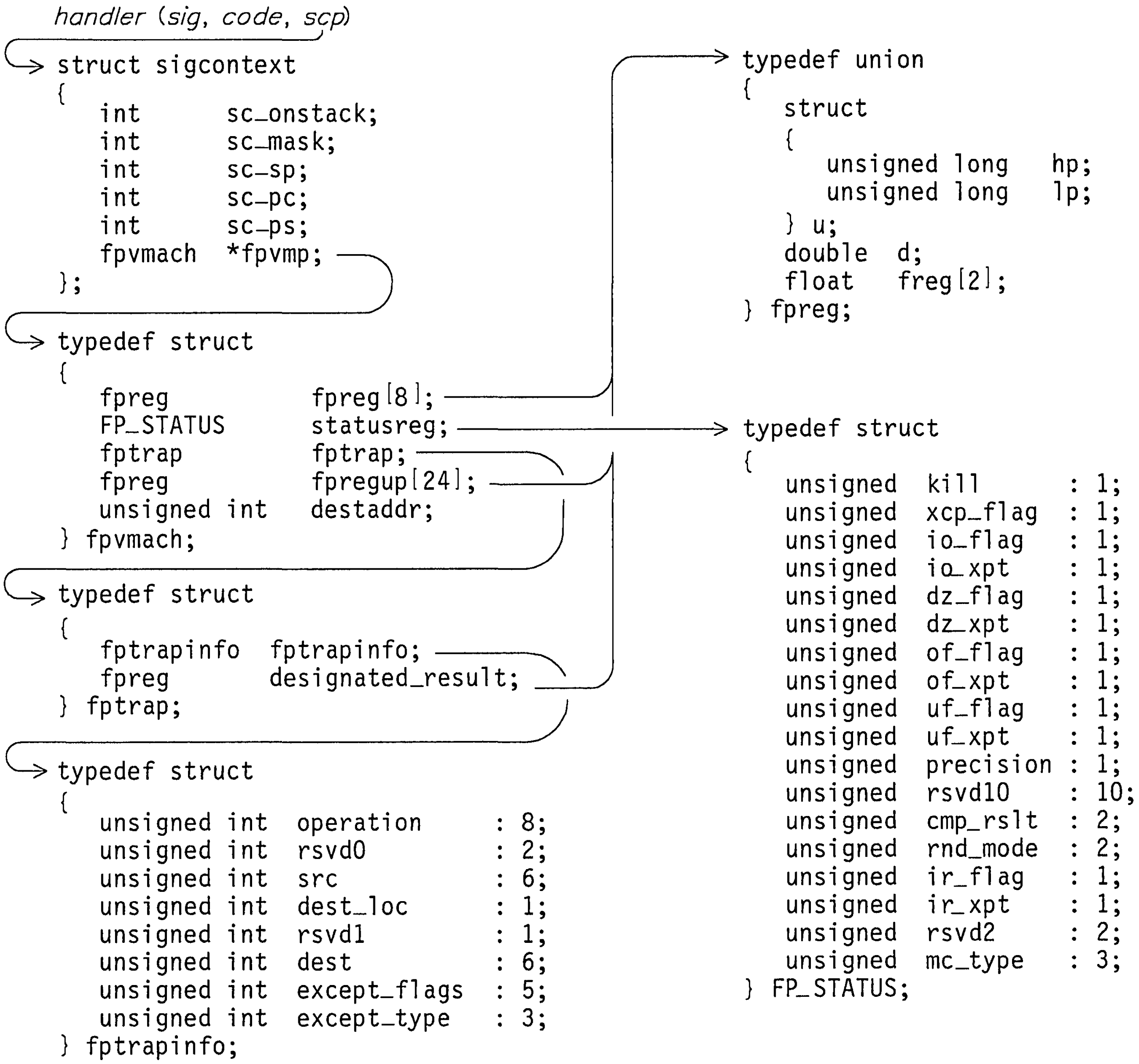
`fpreg`	`fpreg[8]`	Floating-point registers 0-7
`FP_STATUS`	`statusreg`	Status register at time of exception
`fptrap`	`fptrap`	Specific information about the exception
`fpreg`	`fpregup[24]`	Floating-point registers 8-31
`unsigned int`	`destaddr`	Destination address

Each **fpreg** structure represents a double-precision value, two single-precision values, or two long unsigned integers. Viewing a value as a pair of long unsigned integers can be useful for displaying and studying its bit pattern. Each register structure always contains a double value on systems with the APC/881 but no Advanced Floating-Point Accelerator. The **FP_STATUS** structure is discussed in "Status Register Operations" on page 3-186. The **fptrap** structure, also defined in **ieeetrap.h**, contains the following members:

`fptrapinfo`	`fptrapinfo`
`fpreg`	`designated_result`

The **fptrapinfo** structure contains the following members:

`unsigned int`	`operation`	: 8	Operation causing the exception
`unsigned int`	`src`	: 6	Source register number
`unsigned int`	`dest_loc`	: 1	Destination location
`unsigned int`	`dest`	: 6	Destination register number
`unsigned int`	`except_flags`	: 5	Exceptions that occurred
`unsigned int`	`except_type`	: 3	Type of exception



handler (sig, code, scp)

```
struct sigcontext
{
    int       sc_onstack;
    int       sc_mask;
    int       sc_sp;
    int       sc_pc;
    int       sc_ps;
    fpvmach   *fpvmp;
};

typedef struct
{
    fpreg          fpreg[8];
    FP_STATUS      statusreg;
    fptrap         fptrap;
    fpreg          fpregup[24];
    unsigned int   destaddr;
} fpvmach;

typedef struct
{
    fptrapinfo  fptrapinfo;
    fpreg       designated_result;
} fptrap;

typedef struct
{
    unsigned int  operation      : 8;
    unsigned int  rsvd0          : 2;
    unsigned int  src            : 6;
    unsigned int  dest_loc       : 1;
    unsigned int  rsvd1          : 1;
    unsigned int  dest           : 6;
    unsigned int  except_flags   : 5;
    unsigned int  except_type    : 3;
} fptrapinfo;
```

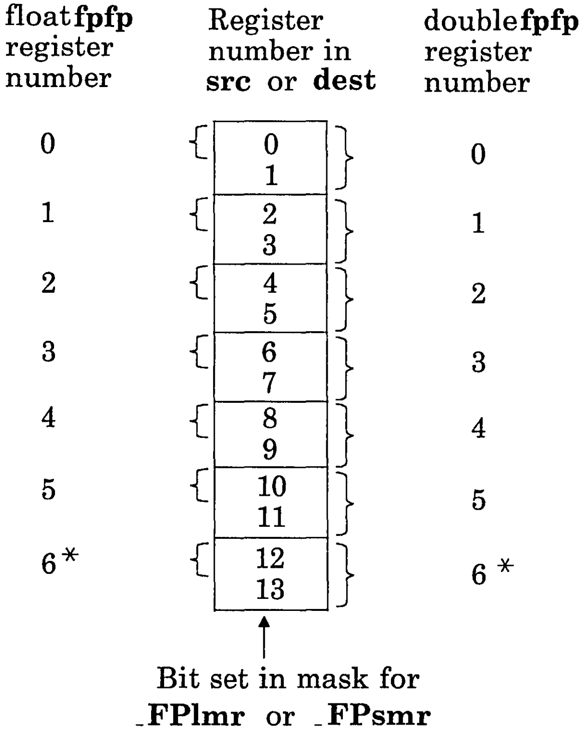
```
typedef union
{
    struct
    {
        unsigned long   hp;
        unsigned long   lp;
    } u;
    double  d;
    float   freg[2];
} fpreg;

typedef struct
{
    unsigned  kill      : 1;
    unsigned  xcp_flag  : 1;
    unsigned  io_flag   : 1;
    unsigned  io_xpt    : 1;
    unsigned  dz_flag   : 1;
    unsigned  dz_xpt    : 1;
    unsigned  of_flag   : 1;
    unsigned  of_xpt    : 1;
    unsigned  uf_flag   : 1;
    unsigned  uf_xpt    : 1;
    unsigned  precision : 1;
    unsigned  rsvd10    : 10;
    unsigned  cmp_rslt  : 2;
    unsigned  rnd_mode  : 2;
    unsigned  ir_flag   : 1;
    unsigned  ir_xpt    : 1;
    unsigned  rsvd2     : 2;
    unsigned  mc_type   : 3;
} FP_STATUS;
```

Figure 3-1. Floating-Point Trap Handler Structures

The **operation** field contains one of the values defined by the **FPFPI** enumeration data type, identifying the operation that caused the exception. See "Table-Driven Interface" on page 3-171 for details about **FPFPI**.

The **src** and **dest** fields specify the operand registers, but not by their abstract **fpfp** register numbers. The operand registers given in **src** and **dest** are numbered from 0 to 13 for single-precision operands. Double-precision operands occupy a pair of registers identified by the even number of the pair. For example, 4 identifies the double-precision value occupying registers 4 and 5, and it corresponds to the abstract **fpfp** register 2. This register mapping is shown in Figure 3-2.



float**fpfp** register number	Register number in **src** or **dest**	double**fpfp** register number
0	0 1	0
1	2 3	1
2	4 5	2
3	6 7	3
4	8 9	4
5	10 11	5
6*	12 13	6 *

* **fpfp** register 6 is a scratch register that is reserved for use by the AIX kernel. User programs should not modify this register.

Figure 3-2. The fpfp Register Mapping

The **dest_loc** field specifies whether the result is a memory location or a floating-point register. If **dest_loc** is set to 0, then the **dest** field specifies the destination register. If **dest_loc** is set to 1, then the **destaddr** field in the **fpvmach** structrue contains the memory address of the destination, and the value of the **dest** field is undefined.

The **except_flags** field indicates the exception or exceptions that resulted from the operation. This value is constructed by logically OR-ing together one or more of **EM_INV_OPER**, **EM_OVERFLOW**, **EM_UNDERFLOW**, **EM_DIVIDE**, and **EM_INEXACT**.

The **except_type** field identifies the exception that caused the trap to be taken. This is the highest-priority exception that occurred for which trapping is enabled. The value is one of **FP_INV_OPER**, **FP_OVERFLOW**, **FP_UNDERFLOW**, **FP_DIVIDE**, or **FP_INEXACT**.

On an inexact, overflow, or underflow exception, the **designated_result** in the **fptrap** structure contains the ***designated result***, as defined by the IEEE standard. It is properly rounded to the precision of the destination of the operation. In the case of overflow or underflow, this result is also scaled.

Warning: When an exception is trapped, the status register is not automatically set to reflect the exception, nor is a result stored in the destination register. The trap handler must do this or the results will be unpredictable.

To set the status register, set the appropriate members of the **fpvmach** structure. That is, set the status by modifying the bit fields of *scp*->**fpvmp**->**statusreg**. To set the result value, do one of the following:

1. If the destination is a register (that is, *scp*->**fpvmp**->**fptrap.fptrapinfo.dest_loc** = = 0), then store the result into *scp*->**fpvmp**->**fpreg[dest]**.

2. If the destination is a memory location (that is, *scp*->**fpvmp**->**fptrap.fptrapinfo.dest_loc** = = 1), then store the result into ***(***scp*->**fpvmp**->**destaddr)**.

Related Information

The description of the Floating-Point Accelerator, Advanced Floating-Point Accelerator, or APC/881 in *Hardware Technical Reference.* The description of the Floating-Point Accelerator or Advanced Floating-Point Accelerator in *Hardware Technical Reference.*

The discussion of Floating-Point Services in *Virtual Resource Manager Technical Reference.*

Assembler Language Reference.

The **cc** command in *AIX Operating System Commands Reference.*

fread, fwrite

Purpose

Performs binary input/output.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int fread ((char *) *ptr*, **sizeof (****ptr***)**, *nitems*, *stream***)**
int *nitems*;
**FILE ****stream*;

int fwrite ((char *) *ptr*, **sizeof (****ptr***)**, *nitems*, *stream***)**
int *size*, *nitems*;
**FILE ****stream*;

Description

The **fread** subroutine copies *nitems* items of data from the input *stream* into an array beginning at the location pointed to by the *ptr* parameter. Each data item has the type **ptr*.

The **fread** subroutine stops copying bytes if an end-of-file or error condition is encountered while reading from the input specified by the *stream* parameter, or when the number of data items specified by the *nitems* parameter have been copied. **fread** leaves the file pointer of *stream*, if defined, pointing to the byte following the last byte read, if there is one. The **fread** subroutine does not change the contents of *stream*.

The **fwrite** subroutine appends *nitems* items of data of the type **ptr* from the array pointed to by the *ptr* parameter to the output *stream*.

The **fwrite** subroutine stops writing bytes if an error condition is encountered on *stream*, or when the number of items of data specified by the *nitems* parameter have been written. The **fwrite** subroutine does not change the contents of the array pointed to by the *ptr* parameter.

Return Value

The **fread** and **fwrite** subroutines return the number of items actually read or written. If the *nitems* parameter is negative or 0, no characters are read or written, and a value of 0 is returned.

Related Information

In this book: “read, readx” on page 2-106, “write, writex” on page 2-184, “fopen, freopen, fdopen” on page 3-168, “getc, fgetc, getchar, getw” on page 3-204, “gets, fgets” on page 3-221, “printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf” on page 3-300, “putc, putchar, fputc, putw” on page 3-309, “puts, fputs” on page 3-313, “scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf” on page 3-325, and “standard i/o library” on page 3-342.

frexp, ldexp, modf

Purpose

Manipulates parts of floating-point numbers.

Library

Standard C Library (**libc.a**)

Syntax

double frexp (*value*, *eptr***)**
double *value*;
int **eptr*;

double ldexp (*mant*, *exp***)**
double *mant*;
int *exp*;

double modf (*value*, *iptr***)**
double *value*, **iptr*;

Description

Every nonzero number can be written uniquely as $x \times 2^n$, where the mantissa (fraction), x, is in the range $0.5 \leq |x| < 1.0$, and the exponent, n, is an integer. The internal representation of floating-point numbers uses this fact, storing a mantissa part and an exponent part.

The **frexp** subroutine returns the mantissa of *value* parameter and stores the exponent in the location pointed to by the *eptr* parameter.

The **ldexp** subroutine returns the quantity $mant \times 2^{exp}$.

The **modf** subroutine returns the signed fractional part of the *value* parameter and stores the integral part in the location pointed to by the *iptr* parameter.

If the **ldexp** subroutine overflows, it returns **HUGE** sets **errno** to **ERANGE**.

Related Information

In this book: "sgetl, sputl" on page 3-334.

fseek, rewind, ftell

Purpose

Repositions the file pointer of a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int fseek (*stream*, *offset*, *whence***)**
**FILE ****stream*;
long *offset*;
int *whence*;

void rewind (*stream***)**
**FILE ****stream*;

long ftell (*stream***)**
**FILE ****stream*;

Description

The **fseek** subroutine sets the position of the next input or output operation on the I/O stream specified by the *stream* parameter. The position of the next operation is determined by the *offset* parameter, which can be either positive or negative.

The **fseek** subroutine sets the file pointer associated with the specified *stream* as follows:

- If the *whence* parameter is 0, the pointer is set to the value of the *offset* parameter.
- If the *whence* parameter is 1, the pointer is set to its current location plus the value of the *offset* parameter.
- If the *whence* parameter is 2, the pointer is set to the size of the file plus the value of the *offset* parameter.

The **fseek** subroutine fails if attempted on a file that has not been opened using **fopen**. In particular, **fseek** cannot be used on a terminal, or on a file opened with **popen**.

Upon successful completion, **fseek** returns a value of 0. If **fseek** fails, a nonzero value is returned.

The **rewind** subroutine is equivalent to **fseek (***stream*, **(long) 0, 0)**, except that it does not return a value.

The **fseek** and **rewind** subroutines undo any effects of the **ungetc** subroutine.

After an **fseek** or a **rewind**, the next operation on a file opened for update can be either input or output.

The **ftell** subroutine returns the offset of the current byte relative to the beginning of the file associated with the named *stream*.

Related Information

In this book: "lseek" on page 2-67, "fopen, freopen, fdopen" on page 3-168, and "standard i/o library" on page 3-342.

ftok

Purpose

Generates a standard interprocess communication key.

Library

Standard C Library (**libc.a**)

Syntax

#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok (*path*, *id***)**
char ***path*;**
char *id***;**

Description

The **ftok** subroutine returns a key, based on the *path* and *id* parameters, to be used to obtain interprocess communication identifiers. The *path* parameter must be the path name of an existing file that is accessible to the process. The *id* parameter must be a character that uniquely identifies a project. **ftok** returns the same key for linked files if called with the same *id* parameter. Different keys are returned for the same file if different *id* parameters are used.

All interprocess communication facilities require you to supply a key to the **msgget**, **semget**, and **shmget** system calls in order to obtain interprocess communication identifiers. The **ftok** subroutine provides one method of creating keys, but many others are possible. Another way to do this, for example, is to use the project ID as the most significant byte of the key, and to use the remaining portion as a sequence number.

Warning: It is important for each installation to define standards for forming keys. If some standard is not adhered to, then it is possible for unrelated processes to interfere with each other's operation.

Warning: If the *path* parameter of the **ftok** subroutine names a file that has been removed while keys still refer it, then the **ftok** subroutine returns an error. If that file is then recreated, the **ftok** subroutine will probably return a different key than the original one.

Return Value

Upon successful completion, the **ftok** subroutine returns a key that can be passed to the **msgget**, **semget**, or **shmget** system call. The **ftok** subroutine returns **(key_t) -1** if one or more of the following are true:

- The file named by the *path* parameter does not exist.
- The file named by the *path* parameter is not accessible to the process.
- The *id* parameter is 0 ('\0').

Related Information

In this book: "msgget" on page 2-76, "semget" on page 2-119, and "shmget" on page 2-140.

ftw

Purpose

Walks a file tree.

Library

Standard C Library (**libc.a**)

Syntax

#include <ftw.h>

int ftw (*path*, *fn*, *depth***)**
char ****path*;**
int (**fn***) ( );**
int *depth***;**

Description

The **ftw** subroutine recursively searches the directory hierarchy that descends from the directory specified by the *path* parameter.

For each file in the hierarchy, the **ftw** subroutine calls the function specified by the *fn* parameter, passes it a pointer to a null-terminated character string containing the name of the file, a pointer to a **stat** structure containing information about the file, and an integer. (For information about the **stat** structure, see "stat.h" on page 5-69.)

The integer passed to *fn* identifies the file type, and it has one of the following values:

FTW_F	Regular file
FTW_D	Directory
FTW_DNR	Directory that cannot be read
FTW_NS	A file for which **stat** could not be executed successfully.

If the integer is **FTW_DNR**, then the files and subdirectories contained in that directory are not processed.

If the integer is **FTW_NS**, then the **stat** structure contents are meaningless. An example of an file that causes **FTW_NS** to be passed to *fn* is a file in a directory for which you have read permission but not execute (search) permission.

The **ftw** subroutine finishes processing a directory before processing any of its files or subdirectories.

The **ftw** subroutine continues the search until the directory hierarchy specified by the *path* parameter is completed, an invocation of the function specified by the *fn* parameter returns a nonzero value, or an error is detected within **ftw**, such as an I/O error.

If the directory hierarchy is completed, the **ftw** subroutine returns a value of 0. If the function specified by the *fn* parameter returns a nonzero value, **ftw** stops its search and returns the value that was returned by the function. If the **ftw** subroutine detects an error, a value of -1 is returned and **errno** is set to indicate the error.

The **ftw** subroutine uses one file descriptor for each level in the tree. The *depth* parameter specifies the maximum number of file descriptors to be used. In general, the **ftw** subroutine runs faster if the value of the *depth* parameter is at least as large as the number of levels in the tree. However, the *depth* parameter must not be greater than the number of file descriptors currently available for use. If the value of the *depth* parameter is 0 or negative, the effect is the same as if it were 1.

Because the **ftw** subroutine is recursive, it is possible for it to terminate with a memory fault due to stack overflow when applied to very deep file structures.

The **ftw** subroutine uses the **malloc** subroutine to allocate dynamic storage during its operation. If **ftw** is terminated prior to its completion, such as by **longjmp** being executed by the function specified by the *fn* parameter or by an interrupt routine, then *ftw* cannot free that storage. The storage remains allocated. A safe way to handle interrupts is to store the fact that an interrupt has occurred, and arrange to have the function specified by the *fn* parameter return a nonzero value the next time it is called.

Related Information

In this book: "signal" on page 2-145, "malloc, free, realloc, calloc" on page 3-236, "setjmp, longjmp" on page 3-332, and "stat.h" on page 5-69.

gamma

Purpose

Computes the logarithm of the gamma function.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

extern int signgam;

double gamma (*x*)
double *x*;

Description

The **gamma** subroutine returns **ln**($|\Gamma(x)|$), where $\Gamma(x)$ is defined as:

$$\int_0^{\infty} e^{-t}\ t^{x-1}\ dt$$

The sign of $\Gamma(x)$ is stored in the external integer variable **signgam**. The *x* parameter cannot be a nonpositive integer.

If the *x* parameter is a nonpositive integer, **gamma** returns **HUGE**, sets **errno** to **EDOM**, and writes a **DOMAIN** error message to standard error.

If the correct value overflows, **gamma** returns **HUGE** and sets **errno** to **ERANGE**.

You can change the error handling procedures with the **matherr** subroutine.

Examples

The following C program fragment calculates Γ(x) and stores the result in y:

```
errno = 0;
y = gamma(x);
if (errno == 0)
   y = signgam * exp(y);
else
   perror("Error in gamma function");
```

Related Information

In this book: "exp, log, log10, pow, sqrt" on page 3-128 and "matherr" on page 3-238.

getc, fgetc, getchar, getw

Purpose

Gets a character or word from an input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int getc (*stream***)**
FILE **stream*;

int fgetc (*stream***)**
FILE **stream*;

int getchar ( )

int getw (*stream***)**
FILE **stream*;

Description

The **getc** macro returns the next character (byte) from the input specified by the *stream* parameter and moves the file pointer, if defined, ahead one character in *stream*. **getc** is a macro and cannot be used where a subroutine is necessary; for example, a subroutine pointer cannot point to it.

Because it is implemented as a macro, **getc** does not work correctly with a *stream* parameter that has side effects. In particular, the following does not work:

```
getc(*f++)
```

In cases like this, use the **fgetc** subroutine instead.

The **fgetc** subroutine performs the same function as **getc**, but **fgetc** is a genuine subroutine, not a macro. The **fgetc** subroutine runs more slowly than **getc**, but takes less space.

The **getchar** macro returns the next character from the standard input stream, **stdin**. Note that **getchar** is also a macro.

The **getw** subroutine returns the next word (**int**) from the input specified by the *stream* parameter and increments the associated file pointer, if defined, to point to the next word. The size of a word varies from one machine architecture to another. The **getw** subroutine returns the constant **EOF** at end-of-file or when an error occurs. Since **EOF** is a valid integer value, **feof** and **ferror** should be used to check the success of **getw**. The **getw** subroutine assumes no special alignment in the file.

Because of possible differences in word length and byte ordering from one machine architecture to another, files written using **putw** are machine-dependent and may not be readable using **getw** on a different type of processor.

Return Value

These subroutines and macros return the integer constant **EOF** at end-of-file or upon an error.

Related Information

In this book: "feof, ferror, clearerr, fileno" on page 3-165 , "fopen, freopen, fdopen" on page 3-168, "fread, fwrite" on page 3-192, "gets, fgets" on page 3-221, "NLgetctab" on page 3-280, "putc, putchar, fputc, putw" on page 3-309, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325, and "standard i/o library" on page 3-342.

getcwd

Purpose

Gets the path name of the current directory.

Library

Standard C Library (**libc.a**)

Syntax

char *getcwd (*buf*, *size***)**
**char ****buf*;
int *size*;

Description

The **getcwd** subroutine returns a pointer to a string containing the path name of the current directory. The value of the *size* parameter must be at least two greater than the length of the path name to be returned.

If the *buf* parameter is a **NULL** pointer, the **getcwd** subroutine will, using the **malloc** subroutine, obtain the number of bytes of free space as specified by the *size* parameter. In this case, the pointer returned by the **getcwd** subroutine can be used as the parameter in a subsequent call to **free**.

The function is implemented by using **popen** to pipe the output of the **pwd** command into the specified string space.

If the **getcwd** subroutine fails, **NULL** is returned and **errno** is set to indicate the error. The **getcwd** subroutine fails if the *size* parameter is not large enough or if an error occurs in a lower-level function.

Related Information

In this book: "malloc, free, realloc, calloc" on page 3-236 and "popen, pclose" on page 3-298.

The **pwd** command in *AIX Operating System Commands Reference.*

getenv, NLgetenv

Purpose

Returns the value of an environment variable.

Library

Standard C Library (**libc.a**)

Syntax

char *getenv (*name***)**
char ****name*;**

char *NLgetenv (*name***)**
char ****name*;**

Description

The **getenv** subroutine searches the environment list for a string of the form *name=value*. Environment variables are sometimes called shell variables since they are frequently set with shell commands.

The **NLgetenv** normally searches the environment for *name* in the same way as **getenv**, except that special actions may be taken if no environment definition is present for *name* (**NULL** is a valid definition) or if a file meant to supercede the current environment is specified through a call to **NLgetfile**.

Return Value

The **getenv** subroutine returns a pointer to the *value* in the current environment if such a string is present. If such a string is not present, a **NULL** pointer is returned.

When no "override" file is found, **NLgetenv** follows this procedure:

1. The definition of *name* is returned, if a non-null definition for *name* exists in the environment.
2. If no definition (or a null definition) of *name* is found in the environment definitions, the environment variable **NLFILE** is searched for the pathname of a file containing environment definitions. These definitions are of the form *parameter=value*. They are read directly from the environment and interpreted as definitions.
   - If a non-null definition for *name* is found, it is returned.

- If a non-null definition for *name* is found, it is returned.
- If a null definition for *name* is found, or if there is no default value for *name*, **NULL** is returned.

3. If no non-null definition for *name* is found in this file, or if **NLFILE** is not defined in the environment, **NLgetenv** returns a traditional default value; if there is no default value for **name**, **NULL** is returned.

If **NLgetfile** is called with a superceding file parameter before **NLgetenv** is called, the search procedure is different. The "override" file is searched; the environment of the process is not consulted in this operation:

1. If a non-null definition for *name* is found in the file, then **NLgetenv** returns the definition.
2. If *name* is not defined in the file, then:
   - A default definition is returned, if one exists.
   - If no default definition exists, **NLgetenv** returns **NULL**.

Related Information

| In this book: "NLgetfile" on page 3-281,"putenv" on page 3-310.1 , and "environment" on page 5-47.

The **sh** command in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

getgrent, getgrgid, getgrnam, setgrent, endgrent

Purpose

Accesses group file entries.

Library

Standard C Library (**libc.a**)

Syntax

#include <grp.h>

struct group *getgrent ( )

struct group *getgrgid (*gid*)
int *gid*;

struct group *getgrnam (*name*)
char **name*;

void setgrent ( )

void endgrent ( )

Description

The **getgrent**, **getgrgid**, and **getgrnam** subroutines return a pointer to a structure containing the broken-out fields of a line in the **/etc/group** file. The **group** structure is defined in the **grp.h** header file, and it contains the following members:

```
char  *gr_name;    /* The name of the group */
char  *gr_passwd;  /* The encrypted group password */
int   gr_gid;      /* The numerical group ID */
char  **gr_mem;    /* Array of pointers to member names */
```

The **getgrent** subroutine, when first called, returns a pointer to the first group structure in the file. On the next call, it returns a pointer to the next group structure in the file. You can call **getgrent** repeatedly to search the entire file.

The **getgrgid** subroutine searches from the beginning of the file until it finds a numerical group ID matching the *gid* parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getgrnam** subroutine searches from the beginning of the file until it finds a group name matching the ASCII equivalent of the *name* parameter. The subroutine then returns a pointer to the structure in which it was found.

If an end-of-file condition or an error is encountered on reading, these functions return a **NULL** pointer.

The **setgrent** subroutine rewinds the group file to allow repeated searches.

The **endgrent** subroutine closes the group file when processing is complete.

Warning: All information is contained in a static area, so it must be copied if it is to be saved.

File

/etc/group

Related Information

In this book: "getlogin" on page 3-212, "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-219, and "group" on page 4-87.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

getlogin

Purpose

Gets the user's login name.

Library

Standard C Library (**libc.a**)

Syntax

char *getlogin ( )

Description

The **getlogin** subroutine returns a pointer to the login name as found in **/etc/utmp**. Use the **getlogin** subroutine in conjunction with the **getpwnam** subroutine to locate the correct password file entry when the same user ID is shared by several login names.

If the **getlogin** subroutine is called within a process that is not attached to a terminal, it returns a **NULL** pointer. The correct procedure for determining the login name is to call **cuserid**, or to call **getlogin** and if it fails, then to call **getpwuid**.

If the login name is not found, **getlogin** returns a **NULL** pointer.

Warning: The **getlogin** subroutine returns a pointer to a static area that is overwritten by successive calls.

File

/etc/utmp

Related Information

In this book: "cuserid" on page 3-62, "getgrent, getgrgid, getgrnam, setgrent, endgrent" on page 3-210, "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-219, and "utmp, wtmp, .ilog" on page 4-170.

getopt

Purpose

Gets flag letters from the argument vector.

Library

Standard C Library (**libc.a**)

Syntax

int getopt (*argc*, *argv*, *optstring***)**
int *argc*;
char ***argv*;
char **optstring*;

extern char *optarg;

extern int optind;

Description

The **getopt** subroutine returns the next flag letter in the *argv* parameter list that matches a letter in the *optstring* parameter. The **getopt** subroutine is an aid to help programs interpret shell command-line flags that are passed to them.

The *optstring* parameter is a string of recognized flag letters. If a letter is followed by a colon, the flag is expected to take a parameter that may or may not be separated from it by white space. The **optarg** external variable is set to point to the start of the flag's parameter on return from the **getopt** subroutine.

The **getopt** subroutine places the *argv* index of the next argument to be processed in **optind**. **optind** is externally initialized to 1 so that *argv*[0] is not processed.

When all flags have been processed (that is, up to the first nonflag argument), the **getopt** subroutine returns **EOF**. The special flag -- (dash dash) can be used to delimit the end of the flags; **EOF** is returned, and -- is skipped.

The **getopt** subroutine prints an error message on **stderr** and returns `(int) '?'` (question mark) when it encounters a flag letter that is not included in the *optstring* parameter.

Examples

The following code fragment processes the flags for a command that can take the mutually exclusive flags **a** and **b**, and the flags **f** and **o**, both of which require parameters.

```
#include <unistd.h>      /* Needed for access system call constants */

main (argc, argv)
int argc;
char **argv;
{
   int c;
   extern int optind;
   extern char *optarg;
   .
   .
   .
   while ((c = getopt(argc, argv, "abf:o:")) != EOF)
   {
      switch (c)
      {
         case 'a':
            if (bflg)
               errflg++;
            else
               aflg++;
            break;
         case 'b':
            if (aflg)
               errflg++;
            else
               bflg++;
            break;
         case 'f':
            ifile = optarg;
            break;
         case 'o':
            ofile = optarg;
            break;
         case '?':
            errflg++;
      } /* case */

      if (errflg)
      {
         fprintf(stderr, "usage: . . . ");
         exit(2);
      }
   } /* while */

   for ( ; optind < argc; optind++)
   {
      if (access(argv[optind], R_OK))
      {
      .
      .
      .
      }
   } /* for  */
}    /* main */
```

Related Information

The **getopt** command in *AIX Operating System Commands Reference.*

getpass

Purpose

Reads a password.

Library

Standard C Library (**libc.a**)

Syntax

char *getpass (*prompt***)**
char ****prompt*;**

Description

The **getpass** subroutine writes the *prompt* string to standard error output, disables echoing, and reads up to a new-line character or EOF from the file **/dev/tty**.

It returns a pointer to a null-terminated string of no more than 8 characters. This return value points to data that is overwritten by successive calls. If the **/dev/tty** file cannot be opened, a **NULL** pointer is returned.

An interrupt terminates input and sends an interrupt signal to the calling program before returning.

File

/dev/tty

Related Information

In this book: "crypt, encrypt" on page 3-42.

getpw

Purpose

Gets a password file entry, given the user ID.

Library

Standard C Library (**libc.a**)

Syntax

int getpw (*uid*, *buf***)**
int *uid***;**
char ****buf*;**

Description

The **getpw** subroutine is included only for compatibility with prior systems and should not be used unless your program is going to be used with a prior system. See "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-219 and "putpwent" on page 3-312 for subroutines to use instead.

The **getpw** searches the password file for a user ID number that matches the *uid* parameter. When a match is found, **getpw** copies the line of the password file in which the match was found into an array pointed to by the *buf* parameter. The subroutine then returns a value of 0. If a match cannot be found, the subroutine returns a nonzero value.

File

/etc/passwd

Related Information

In this book: "passwd" on page 4-112.

getpwent, getpwuid, getpwnam, setpwent, endpwent

Purpose

Gets a password file entry.

Library

Standard C Library (**libc.a**)

Syntax

#include <pwd.h>

struct passwd *getpwent ( )

struct passwd *getpwuid (*uid*)
int *uid*;

struct passwd *getpwnam (*name*)
char **name*;

void setpwent ( )

void endpwent ( )

Description

The **getpwent**, **getpwuid**, and **getpwnam** subroutines return a pointer to a structure containing the broken-out fields of a line in the **/etc/passwd** file. The **passwd** structure is defined in the **pwd.h** header file, and it contains the following members:

```
char  *pw_name;
char  *pw_passwd;
int   pw_uid;
int   pw_gid;
char  *pw_age;
char  *pw_comment;
char  *pw_etc;
char  *pw_dir;
char  *pw_shell;
```

The **pw_comment** field is unused; the others have meanings described in “passwd” on page 4-112.

The **getpwent** subroutine, when first called, returns a pointer to the first **passwd** structure in the file. On the next call, it returns a pointer to the next **passwd** structure in the file. Successive calls can be used to search the entire file.

The **getpwuid** subroutine searches from the beginning of the file until it finds a numerical user ID matching the *uid* parameter. The subroutine then returns a pointer to the structure in which it was found.

The **getpwnam** subroutine searches from the beginning of the file until it finds a login name matching the *name* parameter. The search is made using ***flattened*** names; the characters of the name searched for are the ASCII equivalent character (see "Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System*). The subroutine then returns a pointer to the structure in which it was found.

If an end-of-file condition or an error is encountered on reading, these functions return a **NULL** pointer.

The **setpwent** subroutine rewinds the password file to allow repeated searches.

The **endpwent** subroutine closes the group file when processing is complete.

Warning: All information is contained in a static area, so it must be copied if it is to be saved.

File

/etc/passwd

Related Information

In this book: "getgrent, getgrgid, getgrnam, setgrent, endgrent" on page 3-210. "getlogin" on page 3-212, and "putpwent" on page 3-312.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

gets, fgets

Purpose

Gets a string from a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

char *gets (*s*)
char *_s_;

char *fgets (*s*, *n*, *stream*)
char *_s_;
int *n*;
FILE *_stream_;

Description

The **gets** subroutine reads characters from the standard input stream, **stdin**, into the array pointed to by the *s* parameter. Data is read until a new-line character is read or an end-of-file condition is encountered. If reading is stopped due to a new-line character, the new-line character is discarded and the string is terminated with a null character.

The **fgets** subroutine reads characters from the data pointed to by the *stream* parameter into the array pointed to by the *s* parameter. Data is read until *n* - 1 characters have been read, until a new-line character is read and transferred to *s*, or until an end-of-file condition is encountered. The string is then terminated with a null character.

Return Value

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a **NULL** pointer is returned. If a read error occurs, a **NULL** pointer is returned. Otherwise, *s* is returned.

Related Information

In this book: "feof, ferror, clearerr, fileno" on page 3-165, "fopen, freopen, fdopen" on page 3-168, "fread, fwrite" on page 3-192, "getc, fgetc, getchar, getw" on page 3-204, "puts, fputs" on page 3-313, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325, and "standard i/o library" on page 3-342.

getuinfo

Purpose

Finds the value associated with a user information name.

Library

Standard C Library (**libc.a**)

Syntax

char *getuinfo (*name***)**
char ***name*;**

Description

The **getuinfo** subroutine searches a user information buffer for a string of the form *name*=*value* and returns a pointer to the *value* substring if *name* is found. **NULL** is returned if *name* is not found.

The user information buffer searched is pointed to by the global variable:

```
    extern char *INuibp;
```

This variable is initialized to **NULL.**

If the variable **INuibp** is **NULL** when the **getuinfo** subroutine is called, the **usrinfo** system call is executed to read user information from the kernel into a local buffer. The address of the buffer is then put into the external variable **INuibp**. The **usrinfo** system call is automatically called the first time the **getuinfo** subroutine is called if the **INuibp** variable has not been set.

Related Information

In this book: "usrinfo" on page 2-176.

getutent

Purpose

Accesses **utmp** file entries.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;
```

Description

The **getutent**, **getutid**, and **getutline** subroutines each return a pointer to a structure of the following type:

```
#define ut_name  ut_user
#define ut_id    ut_line

struct utmp
{
   char    ut_user[8];   /* User name                       */
   char    ut_line[12];  /* Device name (console, lnxx) */
   short   ut_pid;       /* Process ID                      */
   short   ut_type;      /* Type of entry                   */
   struct  exit_status
   {
      short  e_termination;  /* Process termination status      */
      short  e_exit;         /* Process exit status             */
   } ut_exit;                /* The exit status of a DEAD_PROCESS */
time_t    ut_time;     /* Time entry was made */
};
```

The **getutent** subroutine reads the next entry from a **utmp**-like file. If the file is not already open, this subroutine opens it. If the end of the file is reached, **getutent** fails.

If you specify type **RUN_LVL**, **BOOT_TIME**, **OLD_TIME**, or **NEW_TIME** in the *id* parameter, the **getutid** subroutine searches forward from the current point in the **utmp** file until an entry with a **ut_type** matching *id*->**ut_type** is found.

If you specify one of the types **INIT_PROCESS**, **LOGIN_PROCESS**, **USER_PROCESS** or **DEAD_PROCESS** in the *id* parameter, then the **getutid** subroutine returns a pointer to the first entry whose type is one of these four and whose **ut_id** field matches *id*->**ut_id**. If the end of the file is reached without a match, the **getutid** subroutine fails.

The **getutline** subroutine searches forward from the current point in the **utmp** file until it finds an entry of the type **LOGIN_PROCESS** or **USER_PROCESS** that also has a **ut_line** string matching the *line*->**ut_line** parameter string. If the end the of file is reached without a match, the **getutline** subroutine fails.

The **pututline** subroutine writes the supplied *utmp* structure into the **utmp** file. If you have not searched for the proper place in the file using one of the **getut** routines, then the **pututline** subroutine calls **getutid** to search forward for the proper place. It is expected that normally the user of **pututline** searched for the proper entry using one of the **getut** subroutines. If so, **pututline** does not search. If the **pututline** subroutine does not find a matching slot for the entry, it adds a new entry to the end of the file.

The **setutent** subroutine resets the input stream to the beginning of the file. You should do this before each search for a new entry if you want to examine the entire file.

The **endutent** subroutine closes the currently open file.

The **utmpname** subroutine changes the name of the file to be examined from **/etc/utmp** to any other file. The name specified is usually **/usr/adm/wtmp**. If the specified file does not exist, no indication is given. You are not aware of this fact until your first attempt to reference the file. The **utmpname** subroutine does not open the file. It closes the old file, if it is currently open, and saves the new file name.

The most current entry is saved in a static structure. If you desire to make multiple accesses, you must copy or use the structure between each access. The **getutid** and **getutline** subroutines examine the static structure first. If the contents of the static structure match what they are searching for, they do not read the **utmp** file. Therefore, you must fill the static structure with zeroes after each use if you want to use these subroutines to search for multiple occurrences.

If **pututline** finds that it isn't already at the correct place in the file, then the implicit read it performs does not overwrite the contents of the static structure returned by the **getutent**, **getuid**, or **getutline** routine. This allows you to get an entry with one of these subroutines, modify the structure, and pass the pointer back to **pututline** for writing.

These subroutines use buffered standard I/O for input, but **pututline** uses an unbuffered nonstandard write to avoid race conditions between processes trying to modify the **utmp** and **wtmp** files.

Return Value

These subroutines fail and return a **NULL** pointer if a read or write fails due to end-of-file or a permission conflict.

Files

/etc/utmp
/usr/adm/wtmp

Related Information

In this book: "ttyslot" on page 3-368and "utmp, wtmp, .ilog" on page 4-170.

hsearch, hcreate, hdestroy

Purpose

Manages hash tables.

Library

Standard C Library (**libc.a**)

Syntax

#include <search.h>

ENTRY *hsearch (*item*, *action***)**
ENTRY *item*;
ACTION *action*;

int hcreate (*nel***)**
unsigned int *nel*;

void hdestroy ( )

Description

The **hsearch** subroutine is a hash table search routine. It returns a pointer into a hash table that indicates the location of a given entry. The *item* parameter is a structure of the type **ENTRY** as defined in the **search.h** header file. It contains two pointers:

item.**key** Points to the comparison key.
item.**data** Points to any other data be associated with that key.

Pointers to types other than **char** should be cast to pointer-to-character. The *action* parameter is a value of the **ACTION** enumeration type that indicates what is to be done with an entry if it cannot be found in the table:

ENTER Enters the item into the table at the appropriate point. If the table is full, a **NULL** pointer is returned.

FIND Does not enter the item into the table, but returns a **NULL** pointer if the item cannot be found.

The **hsearch** subroutine uses open addressing with a multiplicative hash function.

The **hcreate** subroutine allocates sufficient space for the table. You must call **hcreate** before calling **hsearch**. The *nel* parameter is an estimate of the maximum number of entries that the table contains. Under some circumstances, **hcreate** may actually make the table larger than specified. Upon successful completion, **hcreate** returns 1. **hcreate** returns 0 if it cannot allocate sufficient space for the table.

The **hdestroy** subroutine deletes the hash table. This allows you to start a new hash table since only one table can be active at a time.

Related Information

In this book: "bsearch" on page 3-11, " lsearch" on page 3-234, "string" on page 3-344, and "tsearch, tdelete, twalk" on page 3-364.

hypot

Purpose

Computes the euclidean distance function.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double hypot (*x*, *y*)
double *x*, *y*;

Description

The **hypot** subroutine takes precautions against overflows while computing the value of:

$$\sqrt{x^2 + y^2}$$

If the correct value *does* overflow, then **hypot** returns **HUGE** and sets **errno** to **ERANGE**.

You can change the error-handling procedures by supplying a **matherr** subroutine. See "matherr" on page 3-238 for more information.

Related Information

In this book: "exp, log, log10, pow, sqrt" on page 3-128.

initgroups

Purpose

Initializes group access list.

Library

Standard C Library (**libc.a**)

Syntax

int initgroups (*user*, *basegid***)**
char **user*;
int *basegid*;

Description

The **initgroups** subroutine reads the **/etc/group** file and constructs the group access list for the user whose name is specified by the *user* parameter. The *basegid* parameter is usually the group number from the **/etc/password** file and, it is automatically included in the group list.

Warning: The **initgroups** subroutine uses the **getgrent** subroutines. If the program that invokes **initgroups** uses any of these subroutines, then calling **initgroups** overwrites the static group structure.

Return Value

Upon successful completion, the **initgroups** subroutine returns a value of 0. If the effective user ID of the calling process is not superuser, then **initgroups** returns a value of 1.

File

/etc/group

Related Information

In this book: "getgroups" on page 2-52, "setgroups" on page 2-126, and "getgrent, getgrgid, getgrnam, setgrent, endgrent" on page 3-210,

The **adduser** command in *AIX Operating System Commands Reference.*

l3tol, ltol3

Purpose

Converts between 3-byte integers and long integers.

Library

Standard C Library (**libc.a**)

Syntax

void l3tol (*lp*, *cp*, *n***)**
long **lp*;
char **cp*;
int *n*;

void ltol3 (*cp*, *lp*, *n***)**
char **cp*;
long **lp*;
int *n*;

Description

The **l3tol** subroutine converts a list of *n* 3-byte integers packed into a character string pointed to by the *cp* parameter into a list of long integers pointed to by the *lp* parameter.

The **ltol3** subroutine performs the reverse conversion, from long integers (*lp*) to 3-byte integers (*cp*).

These functions are useful for file system maintenance where the block numbers are 3 bytes long.

Warning: The numerical values of the long integers are machine-dependent because of possible differences in byte ordering.

Related Information

In this book: "fs" on page 4-74.

logname

Purpose

Returns the login name of the user.

Library

Programmers Workbench Library (**libPW.a**)

Syntax

char *logname ( )

Description

The **logname** subroutine returns a pointer to the null-terminated login name. The **logname** subroutine extracts the **LOGNAME** variable from the user's environment.

Note: The return values point to static data whose content is overwritten by each call. This method of determining a login name is subject to forgery. For better methods, see "cuserid" on page 3-62, "getlogin" on page 3-212, and "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-219.

File

/etc/profile

Related Information

In this book: "profile" on page 4-127 and "environment" on page 5-47.

The **env** and **login** commands in *AIX Operating System Commands Reference.*

lsearch, lfind

Purpose

Performs a linear search and update.

Library

Standard C Library (**libc.a**)

Syntax

char *lsearch ((char *)*key*, **(char *)***base*, *nelp*, **sizeof (***key*), *compar*)
unsigned int **nelp*;
int (*compar*) **( );**

char *lfind ((char *)*key*, **(char *)***base*, *nelp*, **sizeof (***key*), *compar*)
unsigned int **nelp*;
int (*compar*) **( );**

Description

The **lsearch** subroutine performs a linear search generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.1, Algorithm S.* It returns a pointer into a table indicating where a datum can be found. If the datum does not occur, it is added at the end of the table.

The *key* parameter points to the datum to be sought in the table. The *base* parameter points to the first element in the table. The *nelp* parameter points to an integer containing the current number of elements in the table. This integer is incremented if the datum is added to the table. The *compar* parameter is the name of the comparison function that you must supply (**strcmp**, for example). It is called with two parameters that point to the elements being compared. The *compar* function must return a value of 0 if the elements are equal and nonzero if they are not equal.

The **lfind** subroutine is identical to **lsearch**, except that if the datum is not found, then it is not added to the table. Instead, a **NULL** pointer is returned in this case.

* Reading, Massachusetts: Addison-Wesley, 1981.

The pointers to the key and the element at the base of the table should be of type pointer-to-element and cast to type pointer-to-character. Although it is declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

The comparison function need not compare every byte; therefore, the elements can contain arbitrary data in addition to the values being compared.

Warning: Undefined results can occur if there is not enough room in the table for **lsearch** to add a new item.

Example

The following code fragment reads up to TABSIZE strings, each of which is up to ELSIZE bytes long, and stores them into a table, eliminating duplicates.

```
#include <stdio.h>

#define TABSIZE 50
#define ELSIZE  120

char *lsearch();
int  strcmp();
char line[ELSIZE], tab[TABSIZE][ELSIZE];
unsigned nel = 0;
 . . .
   while (fgets(line, ELSIZE, stdin) != NULL && nel < TABSIZE)
   {
      (void) lsearch(line, (char *)tab, &nel, ELSIZE, strcmp);
   }
 . . .
```

Related Information

In this book: "bsearch" on page 3-11, "hsearch, hcreate, hdestroy" on page 3-227, and "tsearch, tdelete, twalk" on page 3-364.

malloc, free, realloc, calloc

Purpose

Provides a memory allocator.

Library

Standard C Library (**libc.a**)

Syntax

char *malloc (*size***)**
unsigned int *size***;**

void free (*ptr***)**
char ***ptr*;**

char *realloc (*ptr***,** *size***)**
char ***ptr*;**
unsigned int *size***;**

char *calloc (*nelem***,** *elsize***)**
unsigned int *nelem***,** *elsize***;**

Description

The **malloc** and **free** subroutines provide a simple general-purpose memory allocation package.

The **malloc** subroutine returns a pointer to a block of memory of at least *size* bytes. The block is aligned so that it can be used for any type of data. Undefined results occur if the space assigned by **malloc** is overrun.

The **malloc** subroutine searches memory for the first contiguous area of free space of at least *size* bytes. The search is performed in a circular pattern from the last block of memory allocated or freed. During the search, **malloc** joins adjacent free blocks of memory. If a large enough contiguous area of free space is not found, then **malloc** issues a **sbrk** system call to get more memory from the system.

The **free** subroutine frees the block of memory pointed to by the *ptr* parameter for further allocation. The block pointed to by the *ptr* parameter must have been previously allocated by the **malloc** subroutine. The **free** subroutine does not change the contents of this block of memory. Undefined results occur if the *ptr* parameter is not a valid pointer.

The **realloc** subroutine changes the size of the block of memory pointed to by the *ptr* parameter to the number of bytes specified by the *size* parameter, and then it returns a pointer to the block. The contents of the block remain unchanged up to the lesser of the old and new sizes. If a large enough block of memory is not available, then **realloc** calls the **malloc** subroutine to enlarge the memory arena, and then moves the data to the new space.

The **realloc** subroutine also works if the *ptr* parameter points to a block freed since the last call to **malloc, realloc**, or **calloc**.

The **calloc** subroutine allocates space for an array with the number of elements specified by the *nelem* parameter. Each element is of the size specified by the *elsize* parameter. The space is initialized to 0's.

Each of the allocation subroutines returns a pointer to space suitably aligned for storage of any type of object. Cast the pointer to the type *pointer-to-element* before using it.

The **malloc, realloc,** and **calloc** subroutines return a **NULL** pointer if there is no available memory or if the memory arena has been corrupted by storing outside the bounds of a block. When this happens, the block pointed to by the *ptr* parameter could be destroyed.

matherr

Purpose

Performs an action when a math subroutine encounters an error.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

int matherr (*excp*)
**struct exception ****excp*;

Description

If **matherr** error handling is in effect, then the math library subroutines call **matherr** when an error is detected. See "exp, log, log10, pow, sqrt" on page 3-128 and "sin, cos, tan, asin, acos, atan, atan2" on page 3-335 about alternative error handling available for these subroutines.

You can override the default error-handling actions by supplying a subroutine of your own in place of the **matherr** subroutine supplied in the math library. To do this, include in your program a subroutine named **matherr** that takes one parameter: a pointer to an **exception** structure. The **exception** structure is defined in the **math.h** header file and it contains the following members:

```
int    type;
char   *name;
double arg1, arg2, retval;
```

The structure member named **type** describes the type of error that occurred. Its value is one of the following constants:

DOMAIN	Domain error
SING	Singularity
OVERFLOW	Overflow

UNDERFLOW	Underflow
TLOSS	Total loss of significance
PLOSS	Partial loss of significance

The **name** member points to a string containing the name of the subroutine that encountered the error. The members **arg1** and **arg2** contain the parameters that were passed to the subroutine. The **retval** member is the value that the math subroutine returns.

All of the math subroutines that call **matherr** do so in ways similar to this:

```
/*
** Set up the exception structure
*/
exc.type = DOMAIN;    /* Type of error         */
exc.name = "pow";     /* Name of subroutine    */
exc.arg1 = x;         /* Arguments to pow(x,y) */
exc.arg2 = y;

if (matherr(&exc) == 0)
{
   /*
   ** matherr returned 0, so perform the
   ** default error-handling procedures
   */
   fprintf(stderr, "pow: DOMAIN error\n");
   exc.retval = 0;
   errno = EDOM;
}

return (exc.retval);
```

Studying this sample shows that the return value from the **matherr** subroutine controls whether or not the math subroutine performs its default error-handling procedures. If **matherr** returns 0, then the default procedures are performed. Note in particular that if you want to specify the value to be returned by the math subroutine, then your **matherr** subroutine must set *excp*->**retval** and return a nonzero value.

If you do not supply your own **matherr** subroutine, then the **matherr** subroutine supplied in the math library is linked into your program. This subroutine does nothing except return the value 0. Because it returns 0, the calling math subroutine then performs its default error-handling procedures. The default procedures are mentioned in the discussion of each math subroutine.

The math library subroutines **atan**, **ceil**, **erf**, **erfc**, **fabs**, **floor**, **fmod**, and **tanh** do not generate any of the error types listed on page 3-238 and therefore do not call **matherr**.

The following table shows the default error-handling procedures for the remaining math library subroutines:

	DOMAIN	SING	OVERFLOW	UNDERFLOW	TLOSS	PLOSS
acos	M,0,D[1]	—	—	—	—	—
asin	M,0,D[1]	—	—	—	—	—
atan2	M,0,D[2]	—	—	—	—	—
cos	—	—	—	—	M,0,R	*,R
cosh	—	—	H,R	—	—	—
exp	—	—	H,R	0,R	—	—
gamma	—	M,H,D[3]	H,R	—	—	—
hypot	—	—	H,R	—	—	—
j0	—	—	—	0,R	M,0,R	—
j1	—	—	—	0,R	M,0,R	—
jn	—	—	—	0,R	—	—
log	M,-H,D[4]	M,-H,D[5]	—	—	—	—
log10	M,-H,D[4]	M,-H,D[5]	—	—	—	—
pow	M,0,D[6]	—	±H,R	0,R	—	—
sin	—	—	—	—	M,0,R	*,R
sinh	—	—	±H,R	—	—	—
sqrt	M,0,D[7]	—	—	—	—	—
tan	—	—	—	—	M,0,R	*,R
y0	M,-H,D[8]	—	-H,R[9]	0,R	M,0,R	—
y1	M,-H,D[8]	—	—	0,R	M,0,R	—
yn	M,-H,D[8]	—	-H,R[9]	0,R	—	—

Figure 3-3. Default Error-Handling Procedures

The following abbreviations are used in the table:

*	As much as possible of the value is returned.
0	Zero is returned.
H	**HUGE** is returned.
-H	**-HUGE** is returned.

±H **HUGE** or **-HUGE** is returned.
M A message is written to **stdout**.
D **errno** is set to **EDOM**.
R **errno** is set to **ERANGE**.

Notes:

[1] Caused by passing **acos** or **asin** a value larger than 1.0.

[2] Caused by trying to calculate **atan2**(0, 0).

[3] Caused by passing **gamma** a nonpositive integer.

[4] Caused by passing **log** or **log10** a negative value.

[5] Caused by trying to calculate **log**(0) or **log10**(0).

[6] Caused by trying to raise a negative number to a noninteger power or 0 to a nonpositive power.

[7] Caused by passing **sqrt** a negative value.

[8] Caused by passing **y0**, **y1**, or **yn** a nonpositive value.

[9] Caused by passing **y0** a very small positive value.

Examples

The following subroutine suggests the kinds of actions that a user-supplied **matherr** subroutine might perform. It is *not* the **matherr** subroutine that is provided in the math library. The supplied **matherr** subroutine merely returns 0.

```
int matherr(x)
register struct exception *x;
{
   switch (x->type)
   {
      case DOMAIN:
      case SING:
         /* Display message and abort */
         fprintf(stderr, "domain error in %s\n", x->name);
         abort();

      case OVERFLOW:
         if (strcmp("exp", x->name) == 0)
         {
            /* If exp, display message & return the argument */
            fprintf(stderr, "exp of %f\n", x->arg1);
            x->retval = x->arg1;
         }
         else
            if (strcmp("sinh", x->name) == 0)
            {
               /* If sinh, set errno, return 0 */
               errno = ERANGE;
               x->retval = 0;
            }
            else
               /* Otherwise, return HUGE */
               x->retval = HUGE;
         break;

      case UNDERFLOW:
         return (0); /* Perform the default procedures */

      case TLOSS:
      case PLOSS:
         /* Display message and return 0 */
         fprintf(stderr, "loss of significance in %s\n",
            x->name);
         x->retval = 0;
         break;
   } /* switch  */

   return (1); /* Do NOT perform the default procedures */
}    /* matherr */
```

Related Information

In this book: "math.h" on page 5-60.

mdverify

Purpose

Controls write-verify operation for a minidisk.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <mdverify.h>

int mdverify (*md*, *req***)**
char ****md*;**
char *req***;**

Description

The **mdverify** subroutine turns write-verify operation on or off for a given minidisk. When write-verify operation is turned on, the system checks each write operation to the minidisk. After data is written, the system reads it and compares it with the data in the write buffer. If an uncorrectable error is detected, then an error code is passed back from the write operation.

The *md* parameter is a string that specifies the name of the minidisk as it appears in the **/etc/system** file (for example, `"hd1"`). The *req* parameter is one of the following values:

`'v'` Turns write-verify operation on.
`'o'` Turns write-verify operation off.
`'q'` Queries the current write-verify status.

Return Value

Upon successful completion of a `'v'` or `'o'` request, the **mdverify** subroutine returns a value of **MDV_succ**. A successful `'q'` request returns one of the following values:

MDV_wvon Write-verify operation is currently turned on.
MDV_wvoff Write-verify operation is currently turned off.

If the **mdverify** subroutine fails, then it returns one of the following values:

-1 The error is indicated by the value of **errno**.

MDV_open The **/etc/system** file could not be opened.

MDV_nosd The *md* parameter does not specify a valid minidisk name.

MDV_iarg The *req* parameter is not valid.

MDV_csf The IODN could not be found, indicating that the **/etc/system** file has probably been damaged.

MDV_iiodn The minidisk manager detected an invalid minidisk IODN.

MDV_dioe The minidisk manager encountered a disk I/O error.

MDV_iop The minidisk manager encountered an invalid operation mode.

File

/etc/system

Related Information

In this book: "system" on page 4-139 and the **VQUERY** and **VCNTRL ioctl** operations described in "hd" on page 6-20.

The **verify** command in *AIX Operating System Commands Reference.*

memccpy, memchr, memcmp, memcpy, memset

Purpose

Performs memory operations.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

Description

The **memory** subroutines operate on memory areas. A memory area is an array of characters bounded by a count, and not terminated by a null character. The **memory** subroutines do not check for the overflow of any receiving memory area. All of the **memory** subroutines are declared in the **memory.h** header file.

The **memccpy** subroutine copies characters from memory area *s2* into memory area *s1.* The **memccpy** subroutine stops after the first character *c* is copied, or after *n* characters have been copied, whichever comes first. **memccpy** returns a pointer to the character after *c* is copied into *s1*, or a **NULL** pointer if *c* is not found in the first *n* characters of *s2.*

The **memchr** subroutine returns a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s*, or a **NULL** pointer if *c* does not occur.

The **memcmp** subroutine lexicographically compares the first *n* characters in memory area *s1* to the first *n* characters in memory area *s2.* **memcmp** uses native character comparison, which may be signed on some machines. The **memcmp** subroutine returns the following values:

Less than 0	If *s1* is less than *s2*
Equal to 0	If *s1* is equal to *s2*
Greater than 0	If *s1* is greater than *s2*.

The **memcpy** subroutine copies *n* characters from memory area *s2* to area *s1*. It returns *s1*.

The **memset** subroutine sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

Warning: Character movement is performed differently in different implementations of these subroutines; thus overlapping moves may yield unexpected results.

Related Information

In this book: "string" on page 3-344 and "swab" on page 3-349.

mktemp

Purpose

Constructs a unique file name.

Library

Standard C Library (**libc.a**)

Syntax

char *mktemp (*template***)**
char ****template*;**

Description

The **mktemp** subroutine replaces the contents of the string pointed to by the *template* parameter with a unique file name.

The string in the *template* parameter must be a file name with six trailing Xs. The **mktemp** subroutine replaces the Xs with a randomly generated character sequence.

Upon successful completion, the **mktemp** subroutine returns the address of the string pointed to by the *template* parameter.

If the string pointed to by the *template* parameter contains no Xs, or if **mktemp** is unable to construct a unique file name from the randomly generated character sequence, then the first character of the *template* string is replaced with a null character.

Related Information

In this book: "getpid, getpgrp, getppid" on page 2-54, "tmpfile" on page 3-354, and "tmpnam, tempnam" on page 3-355.

monitor

Purpose

Starts and stops execution profiling.

Library

Standard C Library (**libc.a**)

Syntax

#include <mon.h>

void monitor (*lowpc*, *highpc*, *shortbuff*, *bufsize*, *nfunc***)**
— or —
void monitor (0, 0, *profbuff*, **-1,** *nfunc***)**

int (**lowpc***) ( ), (****highpc***) ( );**
**short ****shortbuff*;
**struct prof ****profbuff*;
int *bufsize*, *nfunc*;

Description

The **monitor** subroutine records a histogram of periodically sampled values of the program counter and counts the number of times certain subroutines are called. The **monitor** subroutine is an interface to the **profil** system call.

Executable programs created with **cc -p** automatically include calls to the **monitor** subroutine. You do not need to call the **monitor** subroutine unless you want fine control over profiling.

If the *bufsize* parameter has any value other than -1, then the parameters to **monitor** are interpreted as shown in the first syntax definition. The *lowpc* parameter specifies the lowest address to be sampled, and the highest address to be sampled is the address just below *highpc*. The *lowpc* parameter cannot be 0 when using the **monitor** subroutine to begin profiling. If **monitor** is called with a *lowpc* value of 0, then monitoring is stopped and the results are written to a file named **mon.out**.

The *shortbuff* parameter points to a user-supplied array of short integers. The number of **shorts** in *shortbuff* is specified by the *bufsize* parameter.

The *nfunc* parameter specifies the maximum number of subroutines whose calls are to be counted. Only calls to functions compiled with the **-p** flag of the **cc** command are recorded.

For the results to be significant, especially for programs with small, heavily-used subroutines, specify a buffer that is no more than a few times smaller than the range of locations sampled.

If *bufsize* has the value -1, then the parameters to **monitor** are interpreted as shown in the second syntax definiton. In this case, the arguments *lowpc* and *highpc* are ignored, *nfunc* retains the same meaning as described above, and *profbuff* points to an array of **prof** structures. The **prof** structure is defined in the **mon.h** header file, and it contains the following members:

```
daddr_t  p_low;
daddr_t  p_high;
short_t  *p_buff;
int_t    p_bufsize;
int_t    p_scale;
```

The **monitor** subroutine ignores the value given in **p_scale** and computes a value for it. The **p_high** members in successive structures must be in ascending sequence. The array of structures is terminated with a structure containing a **p_high** member set to zero.

Use the **prof** command to examine the results after executing your program.

Examples

1. To profile the entire program except for floating-point operations:

   ```
   extern etext;
    . . .
   monitor ((int (*)()) 0x10000000, etext, buf, bufsize, nfunc);
   ```

   The identifier **etext** is the address immediately following the program text. (See "end, etext, edata" on page 3-123 for more information about **etext**.)

2. To profile the entire program, including floating-point operations:

   ```
   extern etext;
    . . .
   monitor ((int (*)()) 0x800, etext, buf, bufsize, nfunc);
   ```

   Note that this samples many more instructions, which decreases the resolution of the histogram.

3. To profile an entire program that includes floating-point operations and a shared library:

   ```
   extern etext;
   struct prof buf[4];
    . . .
   buf[0].p_low  = 0x800          /* floating-point text */
   buf[0].p_high = 0x1D108
   buf[1].p_low  = 0x10000000     /* program text        */
   buf[1].p_high = etext
   buf[2].p_low  = 0x40000000     /* shared library text */
   buf[2].p_high = 0x40030D40
   buf[3].p_low  = 0              /* end of array        */
   buf[3].p_high = 0

   monitor(0, 0, buf, -1, nfunc);
   ```

   The addresses shown for the shared library text may differ from the ones appropriate for a program you write.

   The end of the floating-point text may be different on your system. To determine the correct value to use for `buf[0].p_high` in this example, run the following command:

   ```
   nm -x /unix  |  grep _fpend
   ```

   This command displays the value of the symbol **_fpend**, which marks the end of the floating-point code in the kernel.

4. To stop execution monitoring and write the results to the file **mon.out**:

   ```
   monitor ((int(*)()) 0);
   ```

File

mon.out

Related Information

In this book: "profil" on page 2-99 and "end, etext, edata" on page 3-123.

The **cc** and **prof** commands in *AIX Operating System Commands Reference*.

msghelp

Purpose

Issues help text.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <msg00.h>

int msghelp (*flags*, *compid*, *index* [, *fildes*]**)**
unsigned int *flags*;
char **compid*;
int *index*, *fildes*;

Description

The **msghelp** subroutine retrieves a predefined help description from a message/insert/help file and then constructs and outputs the help text.

The *flags* parameter allows default help attributes to be overridden. All flag bits for attributes you do not want to override must be off. If no attributes are overridden, the help is written to **stderr**. Attribute override flag bits that can be set are:

MSGFLFIL Writes the help text to the file specified by the *fildes* parameter. If this flag is not set, then the help text is written to **stderr**.

There is no specific flag bit defined for suppressing output of the help ID. If you want to suppress the help ID, do not specify the *displayed component ID* and *displayed help ID* fields of the help description in the message/insert/help file. If the help ID is suppressed, then the help text is aligned *fildes* parameter causes the help text to be aligned at the left margin instead of to the right of the help ID. This allows a full 79-character width, but does not provide component and help IDs for referencing an explanation of the help in a reference manual.

The *compid* parameter points to a six-character string that identifies the message/insert/help file where the help control information resides. The *compid* parameter is either:

xxxccc For a component file, where, by convention:

xxx Identifies of the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SY***c*, **syc**, **IB***c*, and **ibc**, where *c* is any alphanumeric character.

ccc Identifies the particular software component.

common For the common message/insert/help file.

The *index* parameter is an index into the file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which help description in the file is to be used.

The *fildes* parameter is an integer file descriptor number indicating the opened file to which the help is to be sent. The *fildes* parameter is used only if the **MSGFLFIL** flag is on.

Return Value

Upon successful completion, a value of 0 is returned. If the **msghelp** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID The *compid* parameter is not six characters long. The request is ignored.

MSG_INDX The *index* parameter is not in the range of 1 to 999. The request is ignored.

MSG_TABP The **MSGFLTAB** flag is on. Since helps cannot reside in a message/insert table, this is not a valid flag for the **msghelp** subroutine. The request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The request is ignored.

MSG_SREG A segment register is not available for mapping a message/insert/help file. The request is ignored.

MSG_COMP The message/insert/help file specified by the *compid* parameter cannot be found. Message Services error message 090-002 is output instead.

MSG_INVL The file specified by the *compid* parameter is not a valid message/insert/help file. Message Services error message 090-002 is output instead.

MSG_MTCH The file specified by the *compid* parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that was specified in the file to the **puttext** command when the component file was built. Message Services error message 090-002 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the *index* parameter. Message Services error message 090-002 is output instead.

MSG_REFN The requested help description is found but the description references another help description (in the same file) as the source of the text. The referenced help description does not exist. Message Services error message 090-002 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

-EACCES Search permission is denied for a directory in the path prefix of the message/insert/help file.

-ENOTDIR A component of the path name of the message/insert/help file is not a directory.

-EMFILE Too many files are open for the process.

One of the following values is returned when an attempt to write to the file specified by the *fildes* parameter fails:

-EBADF The *fildes* parameter does not specify a valid file descriptor that is open for writing.

-EFBIG The file specified by the *fildes* parameter exceeds the maximum file size or file size limit for the process.

Related Information

In this book: "msgimed" on page 3-255, "msgqued" on page 3-259, "msgrtrv" on page 3-263, and "message" on page 4-105.

msgimed

Purpose

Issues an immediate message.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <msg00.h>

int msgimed (*flags*, *compid*, *index* [, *sevcode* [, *errcode* [, *fildes*]]]**)**
unsigned int *flags*;
char **compid*;
int *index*, *sevcode*, *fildes*;
long *errcode*;

Description

The **msgimed** subroutine retrieves a predefined message description from a message/insert table or a message/insert/help file and then constructs the message text and outputs it.

The *flags* parameter allows default message attributes to be overridden. All flag bits for attributes you do not want to override must be off. If no attributes are overridden, a message consisting of a message ID (if defined) and message text is written to **stderr**. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the *compid* parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a message/insert/help file.

MSGFLTIM Includes with the message the time the message was issued. The time is given in 24-hour format. This flag should always be set if the error is logged.

MSGFLSEV Includes a severity code with the message. The severity code value is specified by the *sevcode* parameter.

MSGFLERR Includes an error code with the message. The value of the error code is specified by the *errcode* parameter.

MSGFLFIL Writes the message to the file specified by the *fildes* parameter. If this flag is not set, then the message is written to **stderr**.

There is no specific flag bit defined for suppressing output of the message ID. If you want to suppress the message ID, do not specify the **displayed component ID** and the **displayed message ID** fields of the message description in the message/insert table or the message/insert/help file. Suppression of the message ID for a message output to **stderr** or to the output specified by the *fildes* parameter causes the message to be aligned at the left margin instead of to the right of the message ID. This allows a full 79-character width, but does not provide component and message IDs for referencing an explanation of the message in a reference manual.

The *compid* parameter is either a pointer to a message/insert table or identifies the message/insert/help file. If the **MSGFLTAB** flag is set, then the *compid* parameter is a pointer to a message/insert table where the message description resides. If the **MSGFLTAB** flag is not set, then the *compid* parameter identifies the message/insert/help file where the message description resides. In this case, the *compid* parameter is either:

xxxccc For a component file, where, by convention:

xxx Identifies of the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SY***c*, **syc**, **IB***c*, and **ibc**, where *c* is any alphanumeric character.

ccc Identifies the particular software component.

common For the common message/insert/help file.

The *index* parameter is an index into the message/insert table or the message/insert/help file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which message description is to be used.

The *sevcode* parameter specifies an integer severity code that is output with the message if the **msgflerr** flag is set. The following severity codes have been defined:

MSGSVSYT	System termination
MSGSVAPT	Application termination
MSGSVOPR	Operator-recoverable error
MSGSVAPR	Application-recoverable error.

If the **MSGFLSEV** flag is not set, and if the *errcode* or *fildes* parameters are specified, then a dummy *sevcode* parameter must be used as a place holder.

The *errcode* parameter is a long integer value that represents an error code with six decimal digits. The error code is output with the message only if the **MSGFLERR** flag is set. The two high-order decimal digits contain the origin code; the four low-order digits contain an application-defined error return code. The origin code is one of the following values:

MSGORIND Indeterminate origin.
MSGORVDD Detected in VRM. Indicates a device driver level failure.
MSGORVCK Detected in VRM. Indicates a check parameter detected failure in VRM.
MSGORVSV Detected in VRM. Indicates an SVC handler detected failure in VRM.
MSGORUDD Detected in AIX device driver.
MSGORUKN Detected in AIX kernel.
MSGORSHL Detected in shell command.
MSGORRTS Detected in run-time service or daemon.
MSGORAPP Detected in application above the application program interface.

If the **MSGFLERR** flag is not set, and if the *fildes* parameter is specified, then a dummy *errcode* parameter must be used as a place holder.

The *fildes* parameter is a file descriptor indicating the opened file to which the message is to be sent. The *fildes* parameter is used only if the **msgflfil** flag is set.

Return Value

Upon successful completion, a value of 0 is returned. If the **msgimed** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file.

MSG_CPID The *compid* parameter is not six characters long. The request is ignored.

MSG_INDX The *index* parameter is not in the range of 1 to 999. The request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The request is ignored.

MSG_SREG A segment register is not available for mapping a message/insert/help file. The request is ignored.

MSG_BADP The message/insert table pointer provided does not point to a message/insert table. The request is ignored.

MSG_TABI The message/insert table that is provided does not contain the requested message. The request is ignored.

MSG_COMP The message/insert/help file specified by the *compid* parameter cannot be found. Message Services error message 090-001 is output instead.

MSG_INVL The file specified by the *compid* parameter is not a valid message/insert/help file. Message Services error message 090-001 is output instead.

MSG_MTCH The file specified by the *compid* parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that is specified in the file to the **puttext** command when the component file was built. Message Services error message 090-001 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the *index* parameter. Message Services error message 090-001 is output instead.

MSG_REFN The requested message description is found but the description references another message description (in the same file) as the source of the text. The referenced message description does not exist. Message Services error message 090-001 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

-EACCES Search permission is denied for a directory in the path prefix. The request is ignored.

-ENOTDIR A component of the path prefix is not a directory. The request is ignored.

-EMFILE Too many files are open for the process. The request is ignored.

One of the following values is returned when an attempt to write to the file specified by the *fildes* parameter fails:

-EBADF Not a valid file descriptor open for writing.

-EFBIG The file exceeds the process's file size limit or the maximum file size.

Related Information

In this book: "msghelp" on page 3-252, "msgqued" on page 3-259, "msgrtrv" on page 3-263, and "message" on page 4-105.

msgqued

Purpose

Issues a queued message.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <msg00.h>

int msgqued (*flags*, *compid*, *index* [, *sevcode* [, *errcode*]])
unsigned int *flags*;
char **compid*;
int *index*, *sevcode*;
long *errcode*;

Description

The **msgqued** subroutine retrieves a predefined message description from a message/insert table or a message/insert/help file and then constructs the message text and writes it to the queued message file, **/qmsg**.

The queued message file is installed with the AIX Operating System. After installation, you can change the default size of the queued message file (six 2048-byte blocks) by using an editor to modify the four-digit value between the first two asterisks (*) in the first line of the file. This four-digit value is in units of 2048-byte blocks.

After **/qmsg** reaches the size specified in the first line, each new message added to the queue overlays the oldest message in the file. The message queue is maintained across IPLs.

Queued messages are directed to the console operator and are generally system type messages.

The *flags* parameter allows default message attributes to be overridden. All flag bits for attributes you do not want to override must not be set. If no attributes are overridden, then the message consists of the message ID (if defined), the message text, and the date and time the message was issued. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the *compid* parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a message/insert/help file.

MSGFLSEV Includes a severity code with the message. The severity code value is specified by the *sevcode* parameter.

MSGFLERR Includes an error code with the message. The error code value must be specified by the *errcode* parameter.

The *compid* parameter is either a pointer to a message/insert table or identifies the message/insert/help file. If the **MSGFLTAB** flag is set, then the *compid* parameter points to a message/insert table where the message description resides. If the **MSGFLTAB** flag is not set, the *compid* parameter identifies the message/insert/help file where the message description resides.

The *index* parameter is an index into the message/insert table or message/insert/help file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which message description in the file is to be used.

The *sevcode* parameter specifies an integer severity code that is written with the message if the **MSGFLERR** flag is set. The following severity codes have been defined:

MSGSVSYT	System termination
MSGSVAPT	Application termination
MSGSVOPR	Operator-recoverable error
MSGSVAPR	Application-recoverable error.

If the **msgflsev** flag is not set, and if the *errcode* parameter is specified, then a dummy *sevcode* parameter must be used as a place holder.

The *errcode* parameter is a long integer value that represents an error code with six decimal digits. The error code is output with the message only if the **MSGFLERR** flag is set. The two high-order decimal digits contain the origin code; the four low-order digits contain an application-defined error return code. The possible values for the origin code are listed in the description. The origin code is one of the following values:

MSGORIND	Indeterminate origin.
MSGORVDD	Detected in VRM. Indicates a device driver level failure.
MSGORVCK	Detected in VRM. Indicates a check parameter detected failure in VRM.
MSGORVSV	Detected in VRM. Indicates an SVC handler detected failure in VRM.
MSGORUDD	Detected in AIX device driver.
MSGORUKN	Detected in AIX kernel.
MSGORSHL	Detected in shell command.

MSGORRTS Detected in run-time service or daemon.
MSGORAPP Detected in application above the application program interface.

Return Value

Upon successful completion, a value of 0 is returned. If the **msgqued** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID The *compid* parameter is not six characters long. The **msgqued** request is ignored.

MSG_INDX The *index* parameter is not in the range of 1 to 999. The **msgqued** request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The **msgqued** request is ignored.

MSG_SREG A segment register is not available for mapping a message/insert/help file. The **msgqued** request is ignored.

MSG_BADP The message/insert table pointer provided does not point to a message/insert table. The **msgqued** request is ignored.

MSG_TABI The message/insert table that is provided does not contain the requested message. The **msgqued** request is ignored.

MSG_COMP The message/insert/help file specified by the *compid* parameter cannot be found. Message Services error message 090-001 is output instead.

MSG_INVL The file specified by the *compid* parameter is not a valid message/insert/help file. Message Services error message 090-001 is output instead.

MSG_MTCH The file specified by the *compid* parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that is specified in the file to the **puttext** command when the component file was built. Message Services error message 090-001 is output instead.

MSG_NONE The correct component files are found, but none contain the message description specified by the *index* parameter. Message Services error message 090-001 is output instead.

MSG_REFN The requested message description is found but the description references another message description (in the same file) as the source of the text. The referenced message description does not exist. Message Services error message 090-001 is output instead.

MSG_EXEC The **fork** or **exec** system call failed while attempting to run the program that updates the queued message file. The **msgqued** request is ignored. The failure of **exec** is not detected if the calling process catches the **SIGCLD** signal. See "signal" on page 2-145 about catching signals and "Special Signals" on page 2-148 about the special handling of **SIGCLD**.

MSG_QMSG The queued message file, **/qmsg**, cannot be opened, or its format is not valid. The **msgqued** request is ignored. This condition is not detected if the calling process catches the **SIGCLD** signal. See "signal" on page 2-145 about catching signals and "Special Signals" on page 2-148 about the special handling of **SIGCLD**.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

-EACCES Search permission is denied for a directory in the path prefix of the message/insert/help file.

-ENOTDIR A component of the path name of the message/insert/help file is not a directory.

-EMFILE Too many files are open for the process.

File

/qmsg

Related Information

In this book: "msghelp" on page 3-252, "msgimed" on page 3-255, "msgrtrv" on page 3-263, and "message" on page 4-105.

msgrtrv

Purpose

Retrieves a message, insert, or help text.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <msg00.h>

int msgrtrv (*flags*, *compid*, *index*, *buf*, *nbytes***)**
unsigned int *flags*, *nbyte*;
char **compid*, **buf*;
int *index*;

Description

The **msgrtrv** subroutine retrieves a predefined message, insert, or help description from a message/insert/help file or a resident message/insert table, and then constructs the message, insert, or help text as specified and returns the text.

The *flags* parameter allows default attributes to be overridden. All flag bits for attributes you do not want to override must not be set. If no attributes are overridden, insert text is retrieved from a file. Attribute override flag bits that can be set are:

MSGFLTAB Indicates that the *compid* parameter is a pointer to a message/insert table instead of a pointer to a six-character component ID identifying a message/insert/help file. The **MSGFLTAB** flag should not be set if the **MSGFLHLP** flag is set because helps reside only in a message/insert/help file, not in a message/insert table.

MSGFLMSG Retrieves message text instead of insert text.

MSGFLHLP Retrieves help text instead of insert text.

The *compid* parameter is either a pointer to a message/insert table or identifies the message/insert/help file. If the **MSGFLTAB** flag is set, then the *compid* parameter points to a message/insert table where the message or insert description resides. If the **MSGFLTAB** flag is not set, then the *compid* parameter identifies the message/insert/help file where the message, insert, or help description resides. In this case, the *compid* parameter is either:

xxxccc For a component file, where, by convention:

xxx Identifies of the software provider or product. IBM reserves the use of the identifiers **COM**, **com**, **SY***c*, **syc**, **IB***c*, and **ibc**, where *c* is any alphanumeric character.

ccc Identifies the particular software component.

common For the common message/insert/help file.

The *index* parameter is an index into the message/insert table or message/insert/help file specified by the *compid* parameter. The *index* parameter is an integer value from 1 to 999 and identifies which message, insert, or help description in the file or table is to be used.

The *buf* parameter must be either a pointer to a buffer or a pointer to a structure, depending on the value of the *nbyte* parameter.

- If the *nbyte* parameter is greater than 0, then *buf* parameter points to a buffer where the message, insert, or help text is to be stored.
- If the *nbyte* parameter is equal to 0, then the *buf* parameter points to a **msg__rtrv** structure provided by the requesting program. The **msg__rtrv** is defined as a **typedef** in the **msg05.h** header file.

The *nbyte* parameter is either the size of the buffer pointed to by the *buf* parameter, or 0. The buffer size should include space for a terminating null character. If the *nbyte* parameter is 0, a buffer is allocated by the **msgrtrv** subroutine. The buffer pointer (*msgbufp* in the **msg05.h** header file) returned by the **msgrtrv** subroutine should always be inspected by the requesting program after the returned text has been processed. If the inspection finds other than a **NULL** buffer pointer, the buffer should be freed. This should be done regardless of the value of the return code.

Return Value

Upon successful completion, a positive value is returned. If the **msgrtrv** subroutine fails, it returns a negative value that indicates the reason why the text could not be retrieved.

The value returned upon successful completion is the actual length of the constructed text, not including the terminating null character. The following should be noted concerning the length:

- If the *nbyte* parameter was 0 and help text with a title was retrieved, the length returned is the sum of the title length and the text length, including the null terminators after the title and the text.
- If the *nbyte* parameter was not 0, and the retrieved text is longer than the buffer provided (minus 1 character for the null terminator), the excess text is truncated. The length of the truncated text is included in the length returned. If the return code value

is greater than the length specified by the *nbyte* parameter minus 1, the following considerations should be noted:

- The length of the text returned in the buffer is the length specified by the *nbyte* parameter minus one instead of the return code value.
- The requesting program knows that the retrieved text had to be truncated in order to fit into the buffer provided.

If the **msghelp** subroutine fails, then it returns one of the following negative values.

The following values are defined in the **msg04.h** header file, which is included by the **msg00.h** header file:

MSG_CPID The *compid* parameter is not six characters long. The request is ignored.

MSG_INDX The *index* parameter is not in the range of 1 to 999. The request is ignored.

MSG_TABP Both the **msgfltab** and **msgflhlp** flags are on. Since helps cannot reside in a message/insert table, this is not a valid combination of flag bits. The request is ignored.

MSG_ALLO The necessary Message Services work area cannot be allocated. The request is ignored.

MSG_SREG A segment register is not available for mapping a message/insert/help file. The request is ignored.

MSG_BADP The message/insert table pointer provided does not point to a message/insert table. The request is ignored.

MSG_TABI The message/insert table that is provided does not contain the requested message or insert. The request is ignored.

MSG_COMP The message/insert/help file specified by the *compid* parameter cannot be found. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_INVL The file specified by the *compid* parameter is not a valid message/insert/help file. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_MTCH The file specified by the *compid* parameter does not contain descriptions for the specified component. The first six characters of the component file name must be identical to the six-character component ID that was specified in the file to the **puttext** command when the component file was built. If a message was specified, then Message Services error message

090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_NONE The correct component files are found, but none contain the message, insert, or help description specified by the *index* parameter. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, then the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

MSG_REFN The requested message, insert, or help description is found but the description references another message, insert, or help description (in the same file) as the source of the text. The referenced message, insert, or help description does not exist. If a message was specified, then Message Services error message 090-001 is output instead. If an insert was specified, the request is ignored. If help text was specified, then Message Services error message 090-002 is output instead.

Note: Certain errors involve the failure of AIX system calls. In these cases, the **msghelp** subroutine negates the error code that the system call stored in **errno** and returns this value.

One of the following values is returned when an attempt to open a message/insert/help file fails:

-EACCES Search permission is denied for a directory in the path prefix of the message/insert/help file.

-ENOTDIR A component of the path name of the message/insert/help file is not a directory.

-EMFILE Too many files are open for the process.

Related Information

In this book: "msghelp" on page 3-252, "msgimed" on page 3-255, "msgqued" on page 3-259, and "message" on page 4-105.

NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol

Purpose

Collates characters for international character support.

Syntax

```
#include <NLchar.h>

int NCcollate (xc)
NLchar xc;

int _NCxcol (index, src, xstr)
int index;
NLchar **src, **xstr;

int _NLxcol (index, src, xstr)
int index;
char **src;
NLchar **xstr;

int NCcoluniq (xc)
NLchar xc;

int NCeqvmap (ucval)
int ucval;
```

Description

The *xc* value is that of an extended character (**NLchar**).

AIX supports a user-configurable collating order per process, using the table file indicated by the **NLCTAB** environment variable. Collating values increment from zero. The **NLcollate** macro, called with an **NLchar**, returns the collating value. **NCcollate** returns a negative value if extended collation applies to the **NLchar**. If ***extended collation*** applies, either the **NLchar** is translated to a different character or string of characters before collation (***1-to-n collation***), or the **NLchar** is to collate as a unit with one or more following **NLchars** (***n-to-1 collation***). For example, the **NLchar** for the code point representing “ö” might translate to the string “oe” before (1-to-*n*) collation or two code points representing “Pi” might translate to a unit “π” before (*n*-to-1) collation.

When **NCcollate** determines that extended collation is required, **_NCxcol** or **_NLxcol** should be called.

The **_NCxcol** subroutine performs extended collation on the following:

index The negative value returned from **NCcollate** that indicates that extended collation is needed.

src A pointer to a string of **NLchar** type.

xstr A pointer to a replacement text string.

- For 1-to-*n* collation, **_NCxcol** writes the address to *xstr* of a replacement string that is interpolated into the collating operation ahead of the remaining text of *src*.
- For *n*-to-1 collation, a null value is written into the pointer.

The *src* string is updated to point past the elements used for extended collation. **_NCxcol** returns -1 if 1-to-*n* collation is required (*xstr* is not null). If *n*-to-1 collation is required, **_NCxcol** returns the collating value of the extended collation.

The **_NLxcol** subroutine performs extended collation on the following:

index The negative value returned from **NCcollate** that indicates that extended collation is needed.

src A pointer to a string of **char** type. The first code point in the string was converted for the preceding call to **NCcollate**.

xstr A pointer to a replacement text string.

- For 1-to-*n* collation, **NLxcol** writes the address to *xstr* of a replacement string that is interpolated into the collating operation ahead of the remaining text of *src*.
- For *n*-to-1 collation, a null value is written into the pointer.

The *src* string is updated to point past the elements used for extended collation. **NLxcol** returns -1 if 1-to-*n* collation is required (*xstr* is not null). If *n*-to-1 collation is required, **NLxcol** returns the collating value of the extended collation.

The **NCcoluniq** macro disables extended collation, simply assigning each **NLchar** a unique value and treating it as a unit. **NCcoluniq** returns its unique collating value, a non-negative integer that does not receive a special interpretation. A context in which **NCcoluniq** might be used is within character ranges in regular expressions.

The **NCeqvmap** macro is a predicate that returns a non-zero value if the corresponding **NLchar** begins an ***equivalence class***, a set of **NLchars** that can be treated as identical in some collating contexts. For example, if any character of an equivalence class is used as the beginning or ending point of a character range, all of the characters in that class are included in the range.

Related Information

The **ctab** command in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NCctype

Purpose

Classify characters for international character support environments.

Syntax

```
#include <NLctype.h>

int NCisNLchar (x)
int x;

int NCisalpha (x)
int x;

int NCisupper (x)
int x;

int NCislower (x)
int x;

int NCisdigit (x)
int x;

int NCisxdigit (x)
int x;

int NCisalnum (x)
int x;

int NCisspace (x)
int x;

int NCispunct (x)
int x;

int NCisprint (x)
int x;

int NCisgraph (x)
int x;

int NCiscntrl (x)
int x;

int NCisshift (x)
int x;
```

Description

Character classification is user-configurable per process, through the table file indicated by the environment variable **NLCTAB**.

These macros classify character-coded integer values using information specified by the current **NLCTAB** file configuration. The parameter *x* is tested as an **NLchar** (an extended character); each macro is a predicate form returning 0 for false, and a non-zero value for true. The value of *x* is in the domain of any legal **NLchar** in a value range from 0 to **NLCHARMAX-1** inclusive, or a special value of -1. If the value of *x* is not in the domain of the macro, the result is undefined.

The **NCisNLchar** macro is defined on all valid integer values, whereas the other macros are defined only where **NCisNLchar** is true, and on the special value of -1 (EOF). See "standard i/o library" on page 3-342.

When a non-zero value is returned for *x*:

NCisNLchar *x* is a valid **NLchar** with a value between 0 and **NLCHARMAX**-1, inclusive.

NCisalpha *x* is an alphabetical character.

NCisupper *x* is an uppercase alphabetical character.

NCislower *x* is a lowercase letter.

NCisdigit *x* is a decimal digit (0-9).

NCisxdigit *x* is a hexadecimal digit (0-9, A-F (or a-f)).

NCisalnum *x* is an alphanumeric character or digit.

NCisspace *x* is a space, tab, carriage return, new-line, vertical tab, or form-feed character.

NCispunct *x* is a punctuation character (neither a control character nor an alphanumeric character).

NCisprint *x* is a printing character (including the space character).

NCisgraph *x* is a printing character, excluding the space character.

NCiscntrl *x* is an ASCII delete character (0177) or an ordinary ASCII control character other than the four single-shift characters.

NCisshift *x* is one of the four single-shift characters that is used as the first byte of an extended character.

Related Information

In this book: "conv" on page 3-39, "ctype" on page 3-49, "getc, fgetc, getchar, getw" on page 3-204, "NLchar" on page 3-276, "standard i/o library" on page 3-342, and "environment" on page 5-47.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NCstring

Purpose

Performs operations on strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

NLchar *NCstrcat (xs1, xs2)
NLchar *xs1, *xs2;

NLchar *NCstrncat (xs1, xs2, n)
NLchar *xs1, *xs2;
int n;

int NCstrcmp (xs1, xs2)
NLchar *xs1, *xs2;

int NCstrncmp (xs1, xs2, n)
NLchar *xs1, *xs2;
int n;

NLchar *NCstrcpy (xs1, xs2)
NLchar *xs1, *xs2;

NLchar *NCstrncpy (xs1, xs2, n)
NLchar *xs1, *xs2;
int n;

int NCstrlen (xs)
NLchar *xs;

NLchar *NCstrchr (xs, x)
NLchar *xs, x;

NLchar *NCstrrchr (xs, x)
NLchar *xs, x;

NLchar *NCstrpbrk (xs1, s2)
NLchar *xs1;
char *s2;

int NCstrspn (xs1, s2)
NLchar *xs1;
char *s2;

int NCstrcspn (xs1, s2)
NLchar *xs1;
char *s2;

NLchar *NCstrtok (xs1, s2)
NLchar *xs1;
char *s2;
```

Description

The **NCstring** subroutines copy, compare, and append strings in memory, and determine such things as location, size, and existence of strings in memory. For these subroutines, a string is an array of **NLchar**s, terminated by a null character. The **NCstring** subroutines parallel the **string** subroutines (see "string" on page 3-344), but operate on strings of type **NLchar** rather than on type **char**, except as specifically noted below.

These subroutines require their parameters (except the *s2* parameter) to be explicitly converted to type **NLchar**, so they should be used on input that is to be scanned many times for each time it is converted. Where this performance concern does not apply, the **NLstring** subroutines are easier to use (see "NLstring" on page 3-285).

The *s2* parameter is a string of type **char** containing code point representations of ASCII characters or extended characters for international character support. This supports the use of a double-quoted string for this parameter in calling programs.

The parameters *xs1*, *xs2* and *s* point to strings of type **NLchar** (arrays of **NLchar**s terminated by a null character). The *s2* parameter points to strings of type **char**.

The subroutines **NCstrcat**, **NCstrncat**, **NCstrcpy**, and **NCstrncpy** all alter *xs1*. They do not check for overflow of the array pointed to by *xs1*. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **NCstrcat** subroutine appends a copy of the string pointed to by the *xs2* parameter to the end of the string pointed to by the *xs1* parameter. The **NCstrcat** subroutine returns a pointer to the null-terminated result.

The **NCstrncat** subroutine copies at most *n* **NLchar**s of *xs2* to the end of the string pointed to by the *xs1* parameter. Copying stops before *n* **NLchar**s if a null character is encountered in the *xs2* string. The **NCstrncat** subroutine returns a pointer to the null-terminated result.

The **NCstrcmp** subroutine lexicographically compares the string pointed to by the *xs1* parameter to the string pointed to by the *xs2* parameter. The **NCstrcmp** subroutine returns a value that is:

Less than 0	If *xs1* is less than *xs2*
Equal to 0	If *xs1* is equal to *xs2*
Greater than 0	If *xs1* is greater than *xs2*.

The **NCstrncmp** subroutine makes the same comparison as **NCstrcmp**, but it compares at most *n* pairs of **NLchars**. Both **NCstrcmp** and **NCstrncmp** use the environment variable **NLCTAB** to determine the collating sequence for performing comparisons. (See "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267 for information on collation for international character support.) Unless a true collating relationship is to be tested for, **strcmp** and **strncmp** can instead be used for equality comparisons. (See "string" on page 3-344.) The bytes will match regardless of the **NLchars** in the string.

The **NCstrcpy** subroutine copies the string pointed to by the *xs2* parameter to the character array pointed to by the *xs1* parameter. Copying stops when the null character is copied. The **NCstrcpy** subroutine returns the value of the *xs1* parameter.

The **NCstrncpy** subroutine copies *n* **NLchar**s from the string pointed to by the *xs2* parameter to the character array pointed to by the *xs1* parameter. If *xs2* is less than *n* **NLchar**s long, then **NCstrncpy** pads *xs1* with trailing null characters to fill *n* **NLchars**. If *xs2* is *n* or more **NLchar**s long, then only the first *n* **NLchars** are copied; the result is not terminated with a null character. The **NCstrncpy** subroutine returns the value of the *xs1* parameter.

The **NCstrlen** subroutine returns the number of **NLchar**s in the string pointed to by the *s* parameter, not including the terminating null character.

The **NCstrchr** subroutine returns a pointer to the first occurrence of the **NLchar** specified by the *x* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the **NLchar** does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NCstrrchr** subroutine returns a pointer to the last occurrence of the character specified by the *x* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the **NLchar** does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NCstrpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *xs1* parameter of any code point from the string pointed to by the *s2* parameter. A **NULL** pointer is returned if no character matches.

The **NCstrspn** subroutine returns the length of the initial segment of the string pointed to by the *xs1* parameter that consists entirely of code points from the string pointed to by the *s2* parameter.

The **NCstrcspn** subroutine returns the length of the initial segment of the string pointed to by the *xs1* parameter that consists entirely of code points *not* from the string pointed to by the *s2* parameter.

The **NCstrtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *xs1* parameter. The *s2* parameter specifies a set of code points as token delimiters. If the *s1* parameter is anything other than **NULL**, then the **NCstrtok** subroutine reads the string pointed to by the *xs1* parameter until it finds one of the delimiter code points specified by the *s2* parameter. It then stores a null character into the string, replacing the delimiter code point, and returns a pointer to the first **NLchar** of the text token. The **NCstrtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *xs1* parameter step through the string. The delimiters specified by the *s2* parameter can be changed for subsequent calls to **NCstrtok.** When no tokens remain in the string pointed to by the *xs1* parameter, the **NCstrtok** subroutine returns a **NULL** pointer.

Related Information

In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267, "NLchar" on page 3-276, "NLstring" on page 3-285, "NLstrtime" on page 3-288, and "string" on page 3-344.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NLchar

Purpose

Handles data type NLchar.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <NLchar.h>

typedef unsigned short NLchar;

int NCdecode (c, x)                       int NCencode (x, c)
char *c;                                  char *x;
NLchar *x;                                char *c;

int NCdecstr (c, x, len)                  int NCencstr (x, c, len)
char *c;                                  NLchar *x;
NLchar *x;                                char *c;
int len;                                  int len;

int NCdec (c, x)                          int NCenc (x, c)
char *c;                                  NLchar *x;
NLchar *x;                                char *c;

int NCdechr (c)                           int NLisNLcp (c)
char *c;                                  char *c;

int NCchrlen (nlchr)                      int NLchrlen (c)
NLchar nlchr;                             char *c;
```

Description

Characters for international character support can be either one or two bytes in length, while all ASCII characters are one byte long. The **NLchar** data type represents both ASCII and extended characters as single units of storage. The **NLchar** subroutines and macros listed here convert between character types **char** and **NLchar** and provide information about a given character of either type.

The **NCdecode** subroutine converts a character starting at *c* into an **NLchar** at *x*, and returns the number of bytes read from *c*. The **NCencode** subroutine makes the inverse translation from type **NLchar** to type **char** and returns the number of bytes written to *c*.

The **NCdecstr** subroutine converts a string of characters from type **char** to type **NLchar**, and the **NCencstr** does the reverse translation. Both subroutines require the address of the source and destination strings and the total number of elements available for the destination string. The destination string terminates with a zero (0) element, which is included in the string length. The destination length should include space for the terminator. If insufficient space is left for the destination string, a portion of it is not converted. The subroutines return the length of the string in elements, including the terminating 0.

The **NCdec** and **NCenc** macros are equivalent to **NCdecode** and **NCencode** respectively. You can use them to avoid the overhead of function calls in situations where the parameters have no side effects.

The **NCdechr** macro is like **NCdecode** except that **NCdechr** simply returns the value of **NLchar** rather than writing the **NLchar** into memory.

The **NLisNLcp**, **NCchrlen**, and **NLchrlen** macros return information about a given character. **NLisNLcp** returns a zero if the character at *c* is not an extended character, but returns the length of the character if it is an extended character. **NCchrlen** returns the length in bytes that an **NLchar** would have if it were converted into an extended or an ASCII character by **NCencode**. **NLcharlen** returns the length in bytes of the extended or ASCII character starting at *c*.

Related Information

In this book: "conv" on page 3-39, "ctype" on page 3-49, and "NCctype" on page 3-270.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NLescstr, NLunescstr, NLflatstr

Purpose

Translates strings of characters.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <ctype.h>

int NLescstr (src, dest, dlen)
char *src, *dest;
int dlen;

int NLflatstr (src, dest, dlen)
char *src, *dest;
int dlen

int NLunescstr (src, dest, dlen)
char *src, *dest;
int dlen;
```

Description

These subroutines use the subroutines described under "conv" on page 3-39 to convert an entire string of type **char**, perhaps containing extended characters, into a string of pure ASCII bytes. Each of these subroutines require three parameters: the *src* address of the source string, the *dest* address of the destination string, and the *dlen* value, giving the total number of bytes available in the destination string. Each writes a result string terminated by a null character and returns its length in bytes. The *dlen* value should include space for the null character. If *dest* is too short to contain the entire output string, not all of *src* is translated.

The **NLescstr** uses the **NCesc** subroutine to translate each ASCII or extended character in *src* to pure ASCII. Each extended character encountered is translated to a printable ASCII escape sequence that uniquely identifies the extended character. See "display symbols" on page 5-24 for a list of these escape sequences.

The **NLunescstr** subroutine performs the inverse translation using the **NCunesc** subroutine to translate each ASCII byte of *src* into *dest*, and translate each ASCII escape sequence back into the extended character it represents.

The **NLflatstr** subroutine uses the **NCflatchr** subroutine to translate each character, ASCII or extended, in *src* to a single ASCII byte in *dest*. The *dest* string may have fewer bytes than the *src* string, but the number of logical characters, or the ***display length***, is the same. See "NLstring" on page 3-285.

Related Information

In this book: "ctype" on page 3-49, "getc, fgetc, getchar, getw" on page 3-204, "NCctype" on page 3-270, "NCstring" on page 3-272, "NLchar" on page 3-276, "NLstring" on page 3-285, and "display symbols" on page 5-24.

NLgetctab

Purpose

Finds and maps character collating and classification tables to code points.

Library

Standard C Library (**libc.a**)

Syntax

void NLgetctab (*ctfile***)**
**char ***ctfile*;

Description

The **NLgetctab** subroutine locates a table file containing character collation and classification information and maps it into memory. AIX provides international character support for character collation and classification in a **ctype** style, that can be user-configured for a process.

If *ctfile* parameter is not a null value, then the file name given is the name of the file used for character collation and classification. If *ctfile* is null, the value of the environment variable **NLCTAB** is used to specify the table file; if **NLCTAB** is undefined or null, the default file **/etc/nls/ctab/default** is used.

The default compilation procedure builds a call to **NLgetctab** into the runtime startup code of a program, so applications do not normally have to call **NLgetctab** explicitly. If an application does not refer to collating and classification information, the startup code reference is satisfied by a dummy module; the table file is only mapped when needed.

Related Information

In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267 and "environment" on page 5-47.

The **ctab** command in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NLgetfile

Purpose

Gets parameter file for international character support.

Syntax

NLgetfile (*filename*)
**char ****filename*;

Description

The **NLgetfile** subroutine lets an application temporarily change the **NLFILE** environment parameters and thereby select the current "language." The **NLFILE** parameters establish environment settings appropriate to a "language" or dialect for the work station.

The *filename* is either **NULL** or points to a character string containing the name of an environment file that should contain definitions in a format required by **NLFILE** to provide international character support:

- If *filename* is **NULL**, international character support is reset to its default. In this mode **NLgetenv** seeks a definition of an international character support environmental variable, first in the process environment and then in the file specified by the environment variable **NLFILE.** If no definition is found, a default value is returned.
- If *filename* is the name of an environment file, the international character support environment variables defined in the file are used instead of those given in the process environment. Neither the process environment nor the file specified by **NLFILE** are used; if an environment variable is not defined in *filename* then a default value is returned. A **NULL** value is returned when no default value exists.

When it succeeds, **NLgetfile** always calls **NLgetctab** to change the the current collating tables. **NLgetctab** must be called after **NLgetfile** to change the current collating table if different tables are desired.

Return Value

When **NLgetfile** succeeds, 0 is returned.

When **NLgetfile** does not succeed, -1 is returned and **errno** is set to indicate the error. **NLgetfile** does not succeed when:

A non-**NULL** *filename* is given and the file cannot be opened or read, or
A **NULL** *filename* is supplied and the **NLFILE** environment variable cannot be opened or read.

Related Information

In this book: "getenv, NLgetenv" on page 3-208, "NLgetctab" on page 3-280, and "environment" on page 5-47.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

nlist

Purpose

Gets entries from a name list.

Library

Standard C Library (**libc.a**)

Syntax

#include <nlist.h>

int nlist (*filename*, *nl*)
char **filename*;
struct nlist **nl*;

Description

The **nlist** subroutine allows a program to examines the name list in the executable file named by the *filename* parameter. It selectively extracts a list of values and places them in the array of **nlist** structures pointed to by the *nl* parameter.

The name list specified by the *nl* parameter consists of an array of structures containing names of variables, types, and values. The list is terminated with an element that has a null string in the **name** structure member. Each variable name is looked up in the name list of the file. If the name is found, the **type** and **value** of the name are inserted in the next two fields. The **type** field is set to 0 unless the file was compiled with the **-g** option. If the name is not found, both the **type** and **value** entries are set to 0.

All entries are set to 0 if the specified file cannot be read or if it does not contain a valid name list.

You can use the **nlist** subroutine to examine the system name list kept in the **/unix** file. By examining this list, you can ensure that your programs obtain current system addresses.

The **nlist.h** header file is automatically included by **a.out.h** for compatibility. However, do not include **a.out.h** if you only need the information necessary to use the **nlist** subroutine. If you do include **a.out.h**, follow the **#include** statement with the line:

```
#undef n_name
```

Return Value

Upon successful completion, a value of 0 is returned. If the **nlist** subroutine fails, a value of -1 is returned.

Related Information

In this book: "a.out" on page 4-5.

The **cc** command in *AIX Operating System Commands Reference.*

NLstring

Purpose

Performs operations on strings containing code points.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

char *NLstrcat (s1, s2)
char *s1, *s2;

char *NLstrncat (s1, s2, n)
char *s1, *s2;
int n;

int NLstrcmp (s1, s2)
char *s1, *s2;

int NLstrncmp (s1, s2, n)
char *s1, *s2;
int n;

char *NLstrcpy (s1, s2)
char *s1, *s2;

char *NLstrncpy (s1, s2, n)
char *s1, *s2;
int n;

int NLstrlen (s)
char *s;

int NLstrdlen (s)
char *s;

char *NLstrchr (s, x)
char *s, x;

char *NLstrrchr (s, x)
char *s, x;

char *NLstrpbrk (s1, s2)
char *s1, *s2;

int NLstrspn (s1, s2)
char *s1, *s2;

int NLstrcspn (s1, s2)
char *s1, *s2;

char *NLstrtok (s1, s2)
char *s1, *s2;
```

Description

The **NLstring** subroutines copy, compare, and append strings in memory, and determine such things as location, size, and existence of strings in memory. A string is an array of code points terminated by a null character. The **NLstring** subroutines parallel the **string** subroutines (see "string" on page 3-344), and **NLstrcat, NLstrncat, NLstrcpy, NLstrncpy**, and **NLstrlen** are identical in function to their **string** counterparts.

The subroutines **NLstrcat**, **NLstrncat**, **NLstrcpy**, and **NLstrncpy** all alter *s1*. They do not check for overflow of the array pointed to by *s1*. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **NLstrcat** subroutine appends a copy of the string pointed to by the *s2* parameter to the end of the string pointed to by the *s1* parameter. The string is at most *n* bytes; this may represent fewer than *n* code points. The **NLstrcat** subroutine returns a pointer to the null-terminated result.

The **NLstrcmp** subroutine lexicographically compares the string pointed to by the *s1* parameter to the string pointed to by the *s2* parameter. The **NLstrcmp** subroutine returns a value that is:

Less than 0	If *s1* is less than *s2*
Equal to 0	If *s1* is equal to *s2*
Greater than 0	If *s1* is greater than *s2*.

The **NLstrncmp** subroutine makes the same comparison as **NLstrcmp**, but it compares at most *n* bytes. Characters that have 2-byte representations can cause **NLstrncmp** to return 0 for unequal strings. If *n* divides a 2-byte character, then the last byte comparison is skipped. If the only difference in the two strings is in that last byte, an incorrect true is returned.

Both the **NLstrcmp** and **NLstrncmp** subroutines use the environment variable **NLCTAB** to determine the collating sequence for performing comparisons. (See "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267 for information on collation for international character support.) Unless a true collating relationship is to be tested for, **strcmp** and **strncmp** can instead be used for equality comparisons. (See "string" on page 3-344.) The bytes will match regardless of code point representations.

The **NLstrcpy** subroutine copies the string pointed to by the *s2* parameter to the character array pointed to by the *s1* parameter, copying at most *n* bytes. If *s2* is shorter than *n*, a null character is added to *s1*. If the length in bytes of *s2* is greater than *n*, the result is not null-terminated. If byte *n* is the first byte of an extended code then byte *n* is not copied; *s1* is *n*-1 in length. The **NLstrcpy** subroutine returns the value of the *s1* parameter.

The **NLstrlen** subroutine returns the number of bytes in the string pointed to by the *s* parameter, not including the terminating null character.

The **NLstrdlen** subroutine returns the number of code points in the string pointed to by *s*, not including the terminating null character.

The **NLstrchr** subroutine returns a pointer to the first occurrence of the code point corresponding to the **NLchar** specified by the *x* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the code point does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NLstrrchr** subroutine returns a pointer to the last occurrence of the code point corresponding to the **NLchar** specified by the *x* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the code point does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **NLstrpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *s1* parameter of any code point from the string pointed to by the *s2* parameter. A **NULL** pointer is returned if no character matches.

The **NLstrspn** subroutine returns the length of the initial segment of the string pointed to by the *s1* parameter that consists entirely of code points from the string pointed to by the *s2* parameter.

The **NLstrcspn** subroutine returns the length of the initial segment of the string pointed to by the *s1* parameter that consists entirely of code points *not* from the string pointed to by the *s2* parameter.

The **NLstrtok** subroutine returns a pointer to an occurrence of text tokens in the string pointed to by the *s1* parameter. The *s2* parameter specifies a set of code points as token delimiters. If the *s1* parameter is anything other than **NULL**, then the **NLstrtok** subroutine reads the string pointed to by the *s1* parameter until it finds one of the delimiter code points specified by the *s2* parameter. It then stores a null character into the string, replacing the delimiter code point, and returns a pointer to the first code point of the text token. The **NLstrtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *s1* parameter step through the string. The delimiters specified by the *s2* parameter can be changed for subsequent calls to **NLstrtok**. When no tokens remain in the string pointed to by the *s1* parameter, the **NLstrtok** subroutine returns a **NULL** pointer.

Related Information

In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267, "NCstring" on page 3-272, "NLchar" on page 3-276, and "string" on page 3-344.

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NLstrtime

Purpose

Formats time and date.

Syntax

int NLstrtime (*str*, *len*, *format*, *tmdate***)**
char **str*, **format*;
int *len*;
struct tm **tmdate*;

Description

The **NLstrtime** subroutine converts the internal time and date specification *tmdate* that is generated by the *localtime* or *gmtime* clock structures of **ctime** (see "ctime, localtime, gmtime, asctime, tzset" on page 3-46) into a character string under the direction of *format.* The resulting string is similar to the result of **printf** *format,* and is placed in the memory location addressed by *str.* It has a maximum length of *len* and terminates with a **NULL**

Many conversion specifications are the same as those used by the **date** command. The interpretation of some conversion specifications is affected by the values of environment variables for international character support (see "environment" on page 5-47).

The *format* parameter is a character string containing two types of objects: plain characters that are simply placed in the output string, and conversion specifications that convert information from *tmdate* into readable form in the output string. Each conversion specification is a sequence of this form:

%[[-]*width*][.*precision*]*type*

- A % (percent sign) introduces a conversion specification.
- An optional decimal digit string specifies a minimum field *width.* A converted value that has fewer characters than the field width is padded with spaces to the right. If the decimal digit string is preceded with a minus sign, padding with spaces occurs to the left of the converted value.

  If no *width* is given, for numeric fields the appropriate default width is used with the field padded on the left with zeros as required. For strings, the output field is made exactly wide enough to contain the string.
- An optional *precision* value gives the maximum number of characters to be printed for the conversion specification. The precision value is a decimal digit string preceded by

a period. If the value to be output is longer than the precision, it is truncated on the right.

- The **type** of conversion is specified by one or two conversion characters. The characters and their meanings are:

  m The month of the year is output as a number between 01 and 12.

  h The short month is output as a string established by the environment variable **NLSMONTH** (`Jan`, for example).

  lh The long month is output as a string established by the environment variable **NLLMONTH** (`January`, for example).

  d The day of the month is output as a number between 01 and 31.

  j The Julian day of the year is output as a number between 001 and 366.

  w The day of the week is output as a number between 0 and 6.

  a The short day of the week is output as a string according to the environment variable **NLDAY** (`Mon`, for example).

  la The long day of the week is output according to the environment variable **NLLDAY** (`Monday`, for example).

  y The year is output as a number between 00 and 99.

  Y The year is output as a number between 0000 and 9999.

  D The date is output in the format specified by the environment variable **NLDATE** (`05/05/86`, for example).

  lD The long date is output in the format specified by the environment variable **NLLDATE** (`Jul 04, 1986`, for example).

  sD The short date is output in the format specified by the (long date) environment variable **NLLDATE**, but the year is omitted (`July 7`, for example).

  H The hour of the day is output as a number between 00 and 23.

  sH The hour of the day is output as a number between 01 and 12.

  M The minute is output as a number between 00 and 59.

  S The second is output as a number between 00 and 59.

  p The AM or PM indicator is output as a string specified by environmental variable **NLTMISC** (`am`, for example).

  z The (standard or daylight-saving) time zone name is output as a string from the environment variable **TZ** (`CDT`, for example).

r The time is output in the format specified by the environment variable **NLtime**, but using a 12 hour clock (`7:07:50 pm`, for example).

T The time is output in the format specified by the environment variable **NLtime** (`19:07:50`, for example).

sT The time is output in the format specified by the environment variable **NLTIME**, but omitting the seconds (`19:07`, for example).

n Only a newline character is output.

t Only a tab character is output.

x Nothing is output; this conversion specification is used only as a delimiter.

% The % (percent) character is output.

Related Information

In this book: "NLtmtime" on page 3-291, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300, and "environment" on page 5-47.

The **date** and **ctime** commands in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

NLtmtime

Purpose

Sets a time structure from string data.

Syntax

#include (time.h)

int NLtmtime (*str*, *format*, *ptm***)**
char **str*, **format*;
struct tm **ptm*;

Description

The **NLtmtime** subroutine sets the fields in the *ptm* time structure with information in a *str* text string that is parsed according to the *format* string. For each field descriptor in the *format* string, data is read from the *str* string and placed into appropriate fields of the *ptm* structure. The *format* string is described by these rules:

- Each field descriptor begins with a % (percent sign).
- A mnemonic string of 1 or 2 characters follows the % sign and indicates the type of field or fields being read.
- A ***blank character*** (tab, space, or newline character) anywhere in the *format* string causes all blank characters at the corresponding location in the *str* string to be skipped.
- Any character in the *format* string that appears in a field descriptor, other than the blank character, must be matched exactly by the same character in the *str* string. If a mismatch occurs, **NLtmtime** stops processing and any information following the mismatch is ignored. The characters and their meanings are:

  m The month of the year is output as a number between 01 and 12.

  h The short month is output as a string established by the environment variable **NLSMONTH** (`Jan`, for example).

  lh The long month is output as a string established by the environment variable **NLLMONTH** (`January`, for example).

  d The day of the month is output as a number between 01 and 31.

  j The Julian day of the year is output as a number between 001 and 366.

w The day of the week is output as a number between 0 and 6.

a The short day of the week is output as a string according to the environment variable **NLDAY** (`Mon`, for example).

la The long day of the week is output according to the environment variable **NLLDAY** (`Monday`, for example).

y The year is output as a number between 00 and 99.

Y The year is output as a number between 0000 and 9999.

D The date is output in the format specified by the environment variable **NLDATE** (`05/05/86`, for example).

lD The long date is output in the format specified by the environment variable **NLLDATE** (`Jul 04, 1986`, for example).

sD The short date is output in the format specified by the (long date) environment variable **NLLDATE**, but the year is omitted (`July 7`, for example).

H The hour of the day is output as a number between 00 and 23.

sH The hour of the day is output as a number between 01 and 12.

M The minute is output as a number between 00 and 59.

S The second is output as a number between 00 and 59.

p The AM or PM indicator is output as a string specified by environmental variable **NLTMISC** (`am`, for example).

z The (standard or daylight-saving) time zone name is output as a string from the environment variable **TZ** (`CDT`, for example).

r The time is output in the format specified by the environment variable **NLtime**, but using a 12 hour clock (`7:07:50 pm`, for example).

T The time is output in the format specified by the environment variable **NLtime** (`19:07:50`, for example).

sT The time is output is output in the format specified by the environment variable **NLTIME**, but omitting the seconds (`19:07`, for example).

The field descriptors are the same as those used by **NLstrtime** except for those that do not specify information.

Related Information

In this book: "ctime, localtime, gmtime, asctime, tzset" on page 3-46, "NLstrtime" on page 3-288, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325, and "environment" on page 5-47.

The **date** and **ctime** commands in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

perror

Purpose

Writes a message explaining a system call error.

Library

Standard C Library (**libc.a**)

Syntax

void perror (*s*)
char *_s_;

extern int errno;
extern char *sys_errlist[ ];
extern int sys_nerr;

Description

The **perror** subroutine writes a message on the standard error output that describes the last error encountered by a system call or library subroutine. The error message includes the parameter string *s* followed by a : (colon), a blank, the message, and a new-line character. To be of the most use, the parameter string *s* should include the name of the program that caused the error. The error number is taken from the external variable **errno**, which is set when an error occurs, but is not cleared when a successful call is made. See Appendix A, "Error Codes" on page A-1 for a discussion of **errno** values and their meanings.

To simplify various message formats, the array of message strings **sys_errlist** is provided. Use **errno** as an index into this table to get the message string without the new-line character. The largest message number provided in the table is **sys_nerr**. Be sure to check **sys_nerr** because new error codes may be added to the system before they are added to the table.

Related Information

In this book: "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300.

plot

Purpose

Performs graphic output.

Library

Graphics Libraries (**libplot.a**, **libprint.a**, **lib300.a**, and others)

Syntax

```
void openpl ( )

void erase ( )

void label (s)
char *s;

void line (x1, y1, x2, y2)
int  x1, y1, x2, y2;

void circle (x, y, r)
int x, y, r;

void arc (x, y, x0, y0, x1, y1)
int x, y, x0, y0, x1, y1;

void move (x, y)
int x, y;

void cont (x, y)
int x, y;

void point (x, y)
int x, y;

void linemod (s)
char *s;

void space (x0, y0, x1, y1)
int x0, y0, x1, y1;

void closepl ( )
```

Description

The **plot** subroutine family generates graphic output in a relatively device-independent manner. The **space** subroutine must be used before any of these functions to declare the amount of space necessary. The **openpl** subroutine must be used before any of the others to open the device for writing. The **closepl** subroutine flushes the output.

The **circle** subroutine draws a circle of radius *r* with center at the point (*x*, *y*).

The **arc** subroutine draws an arc of a circle with center at the point (*x*, *y*) between the points (*x0*, *y0*) and (*x1*, *y1*).

String parameters to the **label** and **linemod** subroutines are terminated by null characters and must not contain new-line characters.

See "plot" on page 4-115 for a description of the effect of the remaining functions.

These routines appear in several separate libraries. The routines in the **libplot.a** library generate device-independent output. The **tplot** command interprets this output for a specific device.

The other versions of these routines each generate output for a specific device. You should normally redirect the output of **libprint.a** to the printer. You can save the output of **libprint.a** in a regular file and print it later. See the **tplot** command in *AIX Operating System Commands Reference* for a description of how to do this.

On an IBM Graphics Printer, the horizontal distance between points is not the same as the vertical distance between points. This means that arcs and circles are drawn as ellipses. Similarly, drawing a square (with four calls to the **line** subroutine) produces a rectangle. To adjust for this, call the **space** subroutine with appropriate scaling factors.

Files

/usr/lib/libplot.a	Produces output for **tplot** filters
/usr/lib/libprint.a	For an IBM PC Graphics Printer
/usr/lib/lib300.a	For DASI 300
/usr/lib/lib300s.a	For DASI 300s
/usr/lib/lib300S.a	For DASI 300S
/usr/lib/lib450.a	For DASI 450
/usr/lib/lib4014.a	For Tektronix 4014

Related Information

In this book: "plot" on page 4-115.

The **graph** and **tplot** commands in *AIX Operating System Commands Reference.*

popen, pclose

Purpose

Initiates a pipe to or from a process.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

FILE *popen (*command*, *type***)**
**char ****command*, **type*;

int pclose (*stream***)**
**FILE ****stream*;

Description

The **popen** subroutine creates a pipe between the calling program and a shell command to be executed.

The *command* parameter points to a null-terminated string containing a shell command line. The *type* parameter pointers to a null-terminated string containing an I/O mode, either "r" for reading or "w" for writing.

The **popen** subroutine returns a pointer to a **FILE** structure for the stream. If the *type* parameter is "r", you can read from the standard output of the command by reading from the file *stream*. If the *type* parameter is "w", you can write to the standard input of the command by writing to the file *stream*.

Use the **pclose** subroutine to close any stream you have opened with the **popen** subroutine. The **pclose** subroutine waits for the associated process to terminate and then returns the exit status of the command.

Because open files are shared, a type "r" command can be used as an input filter and a type "w" as an output filter.

Warning: If the original processes and the process started with **popen** concurrently read or write a common file, neither should use buffered I/O. If they do, the results are unpredictable.

Some problems with an output filter can be prevented by taking care to flush the buffer with the **fflush** subroutine (see "fclose, fflush" on page 3-163).

The **popen** subroutine returns a **NULL** pointer if files or processes cannot be created, or if the shell cannot be accessed.

The **pclose** subroutine returns -1 if *stream* is not associated with a **popen** command.

Related Information

In this book: "pipe" on page 2-95, "wait" on page 2-182, "fclose, fflush" on page 3-163, "fopen, freopen, fdopen" on page 3-168, "standard i/o library" on page 3-342, and "system" on page 3-350.

printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf

Purpose

Prints formatted output.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int printf (*fmt* [, *val*, ...]**)**
char **fmt*;

int fprintf (*stream*, *fmt* [, *val*, ...]**)**
FILE **stream*;
char **fmt*;

int sprintf (*s*, *fmt* [, *val*, ...]**)**
char **s*, **fmt*;

int NLprintf (*fmt* [, *val*, ...]**)**
char **fmt*;

int NLfprintf (*stream*, *fmt* [, *val*, ...]**)**
FILE **stream*;
char **fmt*;

int NLsprintf (*s*, *fmt* [, *val*, ...]**)**
char **s*, **fmt*;

Description

The **printf** subroutine converts, formats, and writes its *val* parameters, under control of the *fmt* parameter, to the standard output stream **stdout**.

The **fprintf** subroutine converts, formats, and writes its *val* parameters, under control of the *fmt* parameter, to the output stream specified by its *stream* parameter.

The **sprintf** subroutine converts, formats, and stores its *val* parameters, under control of the *fmt* parameter, into consecutive bytes starting at the address specified by the *s* parameter. The **sprintf** subroutine places a '\0' (null character) at the end. It is your responsibility to ensure that enough storage space is available to contain the formatted string.

The **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines parallel their corresponding functions, providing conversion types to handle code points and **NLchars**.

The *fmt* parameter is a character string that contains two types of objects:

- Plain characters, which are copied to the output stream.
- Conversion specifications, each of which causes zero or more items to be fetched from the *val* parameter list.

If there are not enough items for the *fmt* in the *val* parameter list, then the results are unpredictable. If more *val*s remain after the entire *fmt* has been processed, they are ignored.

Each conversion specification in the *fmt* parameter has the following syntax:

1. A % (percent) sign.
2. Zero or more *options*, which modify the meaning of the conversion specification. The *option* characters and their meanings are:

   - The result of the conversion is left-justified within the field.

   + The result of a signed conversion always begins with a sign (+ or -).

   blank If the first character of a signed conversion is not a sign, a blank is prefixed to the result. If both the *blank* and + options appear, then the *blank* option is ignored.

   # This option specifies that the value is to be converted to an alternate form. For **c**, **d**, **s**, and **u** conversions, the option has no effect. For **o** conversion, it increases the precision to force the first digit of the result to be a 0. For **x** and **X** conversions, a nonzero result has 0x or 0X prefixed to it. For **e, E, f, g,** and **G** conversions, the result always contains a decimal point, even if no digits follow the decimal point. For **g** and **G** conversions, trailing zeroes are not removed from the result.

   B This option affects conversions using the **s** or **S** conversion characters of the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines only. The **B** flag specifies that *field width* and *precision* are given in bytes rather than in code points.

   N This option affects the **s** and **S** conversion characters of the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines only. The **N** flag specifies that each international character support code point in the converted string converts into a printable ASCII escape sequence that uniquely identifies the code point.

3. An optional decimal digit string that specifies the minimum *field width*. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the left-adjustment option is specified, the field is padded on the right. For the **NLprintf**, **NLfprintf**, and **NLsprintf** subroutines, field width is measured in code points rather than bytes, unless the **B** flag is specified.

4. An optional *precision*. The precision is a **.** (period) followed by a decimal digit string. If no *precision* is given, it is treated as 0. The *precision* specifies:

   - The minimum number of digits to appear for the **d**, **u**, **o**, **x**, or **X** conversions
   - The number of digits to appear after the decimal point for the **e** and **f** conversions
   - The maximum number of significant digits for the **g** conversion
   - The maximum number of characters to be printed from a string in the **s** conversion.

5. An optional **l** (the letter "ell") specifying that a following **d**, **u**, **o**, **x**, or **X** conversion character applies to a **long** integer *val.*

6. A character that indicates the type of conversion to be applied:

   % Performs no conversion. Prints a %.

   d Accepts an integer *val* and converts it to signed decimal notation. The *precision* specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default *precision* is **1**. The result of converting a zero value with a *precision* of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

   u Accepts an integer *value* and converts it to unsigned decimal notation. The *precision* specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default *precision* is **1**. The result of converting a zero value with a *precision* of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

   o Accepts an integer *val* and converts it to octal notation. The *precision* specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default *precision* is **1**. The result of converting a zero value with a *precision* of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

   An octal value for field width is not implied.

   x, X Accepts an integer *val* and converts it to hexadecimal notation. The letters `abcdef` are used for the **x** conversion and the letters `ABCDEF` are used for the **X** conversion. The *precision* specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeroes. The default *precision* is **1**. The result of converting a zero value with a *precision* of zero is a null string. Specifying a field width with a zero as a leading character causes the field width value to be padded with leading zeros.

f Accepts a **float** or **double** *val* and converts it to decimal notation in the format [-]*ddd.ddd*. The number of digits after the decimal point is equal to the *precision* specification. If no *precision* is specified, then six digits are output. If the *precision* is 0, then no decimal point appears.

e, E Accepts a **float** or **double** *val* and converts it to the exponential form [-]*d.ddd***e**±*dd*. There is one digit before the decimal point and the number of digits after the decimal point is equal to the *precision* specification. If no *precision* is specified, then six digits are output. If the *precision* is 0, then no decimal point appears. The **E** conversion character produces a number with **E** instead of **e** before the exponent. The exponent always contains at least two digits.

g, G Accepts a **float** or **double** *val* and converts it in the style of the **e**, **E** or **f** conversion characters, with the *precision* specifying the number of significant digits. Trailing zeroes are removed from the result. A decimal point appears only if it is followed by a digit. The style used depends on the value converted. Style **e** (**E**, if **G** is the flag used) results only if the exponent resulting from the conversion is less than -4, or if it is greater than or equal to the *precision*.

c Accepts and prints the character *val*.

s Accepts a *val* is as a string (character pointer) and characters from the string are printed until a `'\0'` (null character) is encountered or the number of characters indicated by the *precision* is reached. If no *precision* is specified, all characters up to the first null character are printed. If the string pointer *val* has a value of 0 or **NULL**, the results are undefined.

S The corresponding **NLprintf**, **NLfprintf**, or **NLsprintf** *val* is taken to be a pointer to a string of the type **NLchar**. Characters from the string are printed until a \0 (null) character is encountered or the number of characters indicated by *precision* is reached. If no *precision* is specified, all characters up to the first null character are printed. If the string pointer *val* has a value of 0 or **NULL**, the results are undefined.

A *field width* or *precision* may be indicated by an * (asterisk) instead of a digit string. In this case, an integer *val* parameter supplies the field width or precision. The *val* parameter that is converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or *precision* must appear **before** the value (if any) to be converted.

If the result of a conversion is wider than the *field width*, then the field is expanded to contain the converted result. No truncation occurs. However, a small *precision* may cause truncation on the right.

The **e**, **E**, **f** and **g** formats represent the special floating-point values as follows:

Quiet NaN	`+QNaN` or `-QNaN`
Signalling NaN	`+SNaN` or `-SNaN`
±∞	`+INF` or `-INF`
±0	`+0` or `-0`

The representation of the plus sign depends on whether the + or *blank* formatting option is specified.

Return Value

Upon successful completion, each of these subroutines returns the number of display characters in the output string rather than the number of bytes in the string. (The **NLprintf**, **NLfprintf** and **NLsprintf** subroutines use strings that may contain 2-byte **NLchars**.) The value returned by **sprintf** and **NLsprintf** does not include the final `'\0'` character. If an output error occurs, a negative value is returned.

Related Information

In this book: "conv" on page 3-39, "ecvt, fcvt, gcvt" on page 3-121, "putc, putchar, fputc, putw" on page 3-309, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325, and "standard i/o library" on page 3-342.

Examples of using **printf** in *C Language Guide and Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

programmers workbench library

Purpose

Provides subroutines for compatibility with existing programs.

Library

Programmers Workbench Library (**libPW.a**)

Description

The **libpw** subroutines are provided only for compatibility with existing programs. Their use in new programs is not recommended.

alloca (*nbytes***)**
Allocates *nbytes* of automatic memory.

any (*c, s***)**
Determines whether the string *s* contains the character *c*.

anystr (*s1, s2***)**
Determines the offset in string *s1* of the first character that also occurs in string *s2*.

balbrk (*s, open, close, end***)**
Determines the offset in string *s* of the first character in the string *end* that occurs outside of a balanced string as defined by *open* and *close*.

cat (*dest, source1, . . . ,* **0)**
Concatenates the *source* strings and copy them to *dest*.

clean_up ( )
Defaults the cleanup routine.

curdir (*s***)**
Puts the full path name of the current directory in the string *s*.

dname (*p***)**
Determines which directory contains the file *p*.

fatal (*msg***)**
General purpose error handler.

fdfopen (*fd, mode***)**
Same as the **stdio fdopen** subroutine.

giveup (*dump*)
Forces a core dump.

imatch (*pref, s*)
Determines if the string *pref* is an initial substring of the string *s*.

index (*s1, s2*)
Determines the offset of the first occurrence in string *s1* of string *s2*.

lockit (*lockfile, count, pid*)
Creates a lock file.

move (*s1, s2, n*)
Copies the first *n* characters of string *s1* to string *s2*.

patoi (*s*)
Converts string *s* to **int**.

patol (*s*)
Converts string *s* to **long**.

rename (*oldname, newname*)
Renames the file *oldname* to *newname*.

repeat (*dest, s, n*)
Sets *dest* to the string *s* repeated *n* times.

repl (*s, old, new*)
Replaces each occurrence of the character *old* in string *s* with the character *new*.

satoi (*s, ip*)
Converts string *s* to int and save it in **ip*.

setsig ( )
Causes signals to be caught by **setsigl**.

setsigl (*sig*)
General purpose signal handling routine.

sname (*s*)
Gets a pointer to the simple name of full path name *s*.

strend (*s*)
Finds the end of the string *s*.

substr (*s, dest, origin, len*)
Places a substring of string *s* in *dest* using the offset *origin* and the length *len*.

trnslat (*s, old, new, dest*)
Copies string *s* into *dest* and replace any character in *old* with the corresponding characters in *new*.

unlockit (*lockfile, pid*)
Deletes the lock file.

userdir (*uid*)
Gets the user's login directory.

userexit (*code*)
Defaults user exit routine.

username (*uid*)
Gets the user's login name.

verify (*s1, s2*)
Determines the offset in string *s1* of the first character that is not also in string *s2*.

xalloc (*asize*)
Allocates memory.

xcreat (*name, mode*)
Creates a file.

xfree (*aptr*)
Frees memory.

xfreeall ( )
Frees all memory.

xlink (*f1, f2*)
Links files.

xmsg (*file, func*)
Calls the routine **fatal** with an appropriate error message.

xopen (*name, mode*)
Opens a file.

xpipe (*t*)
Creates a pipe.

xunlink (*f*)
Removes a directory entry.

xwrite (*fd, buffer, n*)
Writes *n* bytes to the file associated with *fd* from *buffer*.

zero (*p, n*)
Zeroes *n* bytes starting at address *p*.

zeropad (*s*)
Replaces the initial blanks with the character `'0'` in string *s*.

Related Information

In this book: “logname” on page 3-233 and “regcmp, regex” on page 3-318.

putc, putchar, fputc, putw

Purpose

Writes a character or a word to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int putc(*c***,** *stream***)**
char *c***;**
FILE **stream***;**

int putchar(*c***)**
char *c***;**

int fputc(*c***,** *stream***)**
char *c***;**
FILE **stream***;**

int putw(*w***,** *stream***)**
int *w***;**
FILE **stream***;**

Description

The **putc** macro writes the character *c* to the output specified by the *stream* parameter. The character is written at the position at which the file pointer is currently pointing, if defined.

The **putchar** macro is the same as the **putc** macro except that **putchar** writes to the standard output.

The **fputc** subroutine works the same as **putc**, but **fputc** is a true subroutine rather than a macro. It runs more slowly than **putc**, but takes less space per invocation.

Because **putc** is implemented as a macro, it treats incorrectly a *stream* parameter with side effects, such as `putc(c, *f++)`. For such cases, use **fputc** instead. Also, use **fputc** whenever you need to pass a pointer to this subroutine as a parameter to another subroutine.

The **putw** subroutine writes the word (**int**) specified by the *w* parameter to the output specified by the *stream* parameter. The word is written at the position at which the file pointer, if defined, is pointing. The size of a word is the size of an integer and varies from machine to machine. The **putw** subroutine does not assume or cause special alignment of the data in the file.

Because of possible differences in word length and byte ordering, files written using the **putw** subroutine are machine-dependent, and may not be readable using the **getw** subroutine on a different processor.

With the exception of **stderr**, output streams are, by default, buffered if they refer to files, or line-buffered if they refer to terminals. The standard error output stream, **stderr**, is unbuffered by default, but using the **freopen** subroutine causes it to become buffered or line-buffered. Use the **setbuf** subroutine to change the stream's buffering strategy.

When an output stream is unbuffered, information is queued for writing on the destination file or terminal as soon as it is written. When an output stream is buffered, many characters are saved and written as a block. When an output stream is line-buffered, each line of output is queued for writing on the destination terminal as soon as the line is completed (that is, as soon as a new-line character is written or terminal input is requested).

Return Value

Upon successful completion, these functions each return the value written. If these functions fail, they return the constant **EOF**. They fail if the *stream* is not open for writing, or if the output file size cannot be increased. Because **EOF** is a valid integer, you should use the **ferror** subroutine to detect **putw** errors.

Related Information

In this book: "fclose, fflush" on page 3-163, "feof, ferror, clearerr, fileno" on page 3-165, "fopen, freopen, fdopen" on page 3-168, "fread, fwrite" on page 3-192, "getc, fgetc, getchar, getw" on page 3-204, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300, "puts, fputs" on page 3-313, "setbuf, setvbuf" on page 3-330, and "standard i/o library" on page 3-342.

putenv

Purpose

Sets an environment variable.

Library

None

Syntax

int putenv (*str*)
**char ****str*;

Description

The **putenv** subroutine sets the value of an environment variable by altering an existing variable or by creating a new one. The *str* parameter points to a string of the form *name*=*value*, where *name* is the environment variable and *value* is the new value for it.

The memory space pointed to by the *str* parameter becomes part of the environment, so that altering the string effectively changes part of the environment. The space is no longer used after the the value of the environment variable is changed by calling **putenv** again.

Warning: Unpredictable results can occur if a subroutine passes **putenv** a pointer to an automatic variable and then returns while the variable is still part of the environment.

Note: The **putenv** subroutine manipulates the environment pointed to by the **environ** external variable, and it can be used in conjunction with **getenv**. However, *envp*, the third parameter to **main**, is not changed. See "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34 for more information about **environ** and *envp*.

The **putenv** subroutine uses **malloc** to enlarge the environment.

After **putenv** is called, environment variables are not necessarily in alphabetical order.

Return Value

Upon successful completion, a value of 0 is returned. If **malloc** is unable to obtain sufficient space to expand the environment, then **putenv** returns a nonzero value.

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "getenv, NLgetenv" on page 3-208, "malloc, free, realloc, calloc" on page 3-236, and "environment" on page 5-47.

putpwent

Purpose

Writes a password file entry.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <pwd.h>

int putpwent (*p*, *f*)
**struct passwd ****p*;
**FILE ****f*;

Description

The **putpwent** subroutine writes a line on the stream specified by the *f* parameter. The stream that is written on matches the format of **/etc/passwd**.

The *p* parameter is a pointer to a **passwd** structure created by the **getpwent**, **getpwuid**, or **getpwnam** subroutines.

Return Value

Upon successful completion, **putpwent** returns a value of 0. If **putpwent** fails, a nonzero value is returned.

Related Information

In this book: "getpwent, getpwuid, getpwnam, setpwent, endpwent" on page 3-219 and "passwd" on page 4-112.

puts, fputs

Purpose

Writes a string to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int puts (*s***)** **char ****s***;**	**int fputs (***s*, *stream***)** **char ****s***;** **FILE ****stream***;**

Description

The **puts** subroutine writes the null-terminated string pointed to by the *s* parameter, followed by a new-line character, to the standard output stream, **stdout**.

The **fputs** subroutine writes the null-terminated string pointed to by the *s* parameter to the output stream specified by the *stream* parameter. The **fputs** subroutine does not append a new-line character.

Neither subroutine writes the terminating null character.

Return Value

Upon successful completion, the **puts** and **fputs** subroutines return the number of characters written. Both subroutines return **EOF** on an error. This happens if the routines try to write on a file that has not been opened for writing.

Related Information

In this book: "feof, ferror, clearerr, fileno" on page 3-165, "fopen, freopen, fdopen" on page 3-168, "fread, fwrite" on page 3-192, "gets, fgets" on page 3-221, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300, "putc, putchar, fputc, putw" on page 3-309, and "standard i/o library" on page 3-342.

qsort

Purpose

Sorts a table of data in place.

Library

Standard C Library (**libc.a**)

Syntax

void qsort ((char *) *base*, *nel*, **sizeof (****base*), *compar***)**
unsigned int *nel***;**
int (**compar***) ( );**

Description

The **qsort** subroutine sorts a table of data in place. It uses the "quicker-sort" algorithm.

The *base* parameter points to the element at the base of the table. The *nel* parameter is the number of elements in the table. The *compar* parameter is the name of the comparison function.

The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, *compar* must return a value less than 0.
- If the first parameter is equal to the second parameter, *compar* must return 0.
- If the first parameter is greater than the second parameter, *compar* must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

Note: The order in the output of two items that compare equal is unpredictable.

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character.

Related Information

In this book: "bsearch" on page 3-11, "lsearch" on page 3-234, and "string" on page 3-344.

The **sort** command in *AIX Operating System Commands Reference.*

rand, srand

Purpose

Generates pseudo-random numbers.

Library

Standard C Library (**libc.a**)

Syntax

int rand ( )

void srand (*seed***);**
unsigned int *seed***;**

Description

The **rand** subroutine generates a random numbers using a multiplicative congruential algorithm. The random-number generator has a period of 2^{32}, and it returns successive pseudo-random numbers in the range from 0 to 2^{15} - 1.

The **srand** subroutine resets the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

Note: The **rand** subroutine is a very simple random-number generator. Its spectral properties, the mathematical measurement of how "random" the number sequence is, are somewhat weak. See "drand48" on page 3-118 for a more elaborate random-number generator that has better spectral properties.

regcmp, regex

Purpose

Compiles and matches regular-expression patterns.

Library

Programmers Workbench Library (**libPW.a**)

Syntax

char *regcmp (*str* [, *str*, . . .], **(char *) 0)**
char **str*, **str*, . . . ;

char *regex (*pat*, *subject* [, *ret*, . . .]**)**
char **pat*, **subject*, **ret*, . . . ;

extern char *__locl;

Description

The **regcmp** subroutine compiles a regular expression (or pattern) and returns a pointer to the compiled form. The *str* parameters specify the pattern to be compiled. If more than one *str* parameter is given, then **regcmp** treats them as if they were concatenated together. It returns a **NULL** pointer if it encounters an incorrect parameter.

You can use the **regcmp** command to compile regular expressions into your C program, frequently eliminating the need to call the **regcmp** subroutine at run time.

The **regex** subroutine compares a compiled pattern to the *subject* string. Additional parameters are used to receive values. Upon successful completion, the **regex** subroutine returns a pointer to the next unmatched character. If the **regex** subroutine fails, a **NULL** pointer is returned. A global character pointer, **__locl**, points to where the match began.

The **regcmp** and **regex** subroutines are borrowed from the **ed** command; however, the syntax and semantics have been changed slightly. You can use the following symbols with the **regcmp** and **regex** subroutines:

`[ ] * . ^`
These symbols have the same meaning as they do in the **ed** command.

`-`
For **regex**, the minus within brackets means "through" according to the current collating sequence. For example, `[a-z]` can be equivalent to `[abcd . . . xyz]` or

[aBbCc . . . xYyZz] or even [aàáâbc . . . xyz]. You can use the - by itself if the - is the last or first character. For example, the character class expression []-] matches the] (right bracket) and - (minus) characters.

The **regcmp** subroutine does not use the current collating sequence, and the minus character in brackets controls only a direct ASCII sequence. For example, [a-z] always means [abc . . . xyz] and [A-Z] always means [ABC . . . XYZ]. If you need to control the specific characters in a range using **regcmp**, you must list them explicitly rather than using the minus in the character class expression.

$

Matches the end of the string. Use \n to match a new-line character.

+

A regular expression followed by + means one or more times. For example, [0-9]+ is equivalent to [0-9][0-9]*.

{*m*} {*m*,} {*m*,*u*}

Integer values enclosed in { } indicate the number of times to apply the preceding regular expression. *m* is the minimum number and *u* is the maximum number. *u* must be less than 256. If you specify only *m*, it indicates the exact number of times to apply the regular expression. {*m*,} is equivalent to {*m*,∞} and matches *m* or more occurrences of the expression. The plus + (plus) and * (asterisk) operations are equivalent to {1,} and {0,}, respectively.

(. . .)$*n*

This stores the value matched by the enclosed regular expression in the $(n+1)^{\text{th}}$ *ret* parameter. Ten enclosed regular expressions are allowed. **regex** makes the assignments unconditionally.

(. . .)

Parentheses group subexpressions. An operator, such as *, +, or { } works on a single character or on a regular expression enclosed in parenthesis. For example, (a*(cb+)*)$0.

All of the above defined symbols are special. You must precede them with a \ (backslash) if you want to match the special symbol itself. For example, \$ matches a dollar sign.

Note: regcmp uses the **malloc** subroutine to make the space for the vector. Always free the vectors that are not required. If you do not free the unrequired vectors, you may run out of memory if **regcmp** is called repeatedly. Use the following as a replacement for **malloc** to reuse the same vector, thus saving time and space:

```
/*  . . . Your Program . . .  */

malloc(n)
   int n;
{
   static int rebuf[256];

   return ((n <= sizeof(rebuf)) ? rebuf : NULL);
}
```

Examples

1. To perform a simple match:

   ```
   char *cursor, *newcursor, *ptr;
    . . .
   newcursor = regex((ptr = regcmp("^\n", 0)), cursor);
   free(ptr);
   ```

   This matches a leading new-line character in the subject string pointed to by `cursor`.

2. To extract a substring that matches a pattern:

   ```
   char ret0[9];
   char *newcursor, *name;
    . . .
   name = regcmp("([A-Za-z][A-Za-z0-9]{0,7})$0", 0);
   newcursor = regex(name, "123Testing321", ret0);
   ```

   This matches the eight-character identifier `Testing3` and returns the address of the character after the last matched character (which is stored in `newcursor`). The string `Testing3` is copied into the character array `ret0`.

Related Information

In this book: "malloc, free, realloc, calloc" on page 3-236, "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267, and "regexp: compile, step, advance" on page 3-321.

The **ed** and **regcmp** commands in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

regexp: compile, step, advance

Purpose

Compiles and matches regular-expression patterns.

Library

None

Syntax

```
#define INIT                declarations
#define GETC( )             getc_code
#define PEEKC( )            peekc_code
#define UNGETC(c)           ungetc_code
#define RETURN(pointer)     return_code
#define ERROR(val)          error_code

#include <regexp.h>
```

```
char *compile (instring, expbuf, endbuf, eof)
char *instring, *expbuf, *endbuf;
char eof;
```

```
int step (string, expbuf)
char *string, *expbuf;

int advance (string, expbuf)
char *string, *expbuf;
```

Description

The **regexp.h** header file defines several general-purpose subroutines that perform regular-expression pattern matching. Programs that perform regular-expression pattern matching such as **ed**, **sed**, **grep**, **bs**, and **expr** use this source file. In this way, only this file needs to be changed in order to maintain regular expression compatibility between programs.

The **regexp.h** header file handles extended characters and may require access to the current collating sequence. You can disable the extended functionality of **regexp.h** by defining the preprocessor variable **RTPC_NO_NLS**. This is useful for tasks such as building programs to run on prior releases of AIX. See "Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System* for more information.

The interface to this header file is complex. Programs that include this file define the following five macros before the **#include <regexp.h>** statement. These macros are used by the **compile** subroutine.

INIT

This macro is used for dependent declarations and initializations. It is placed right after the declaration and opening { (left brace) of the **compile** subroutine. The definition of **INIT** must end with a **;** (semicolon). **INIT** is frequently used to set a register variable to point the beginning of the regular expression so that this register variable can be used in the declarations for **GETC**, **PEEKC**, and **UNGETC**. Otherwise, you can use **INIT** to declare external variables that **GETC**, **PEEKC**, and **UNGETC** need.

GETC( )

This macro returns the value of the next character in the regular expression pattern. Successive calls to the **GETC** macro should return successive characters of the pattern.

PEEKC( )

This macro returns the next character in the regular expression. Successive calls to the **PEEKC** macro should return the same character, which should also be the next character returned by the **GETC** macro.

UNGETC(*c*)

This macro causes the parameter *c* to be returned by the next call to the **GETC** and **PEEKC** macros. No more than one character of pushback is ever needed and this character is guaranteed to be that last character read by the **GETC** macro. The return value of the **UNGETC** macro is always ignored.

RETURN(*pointer*)

This macro is used on normal exit of the **compile** subroutine. The *pointer* parameter points to the first character immediately following the compiled regular expression. This is useful to programs that have memory allocation to manage.

ERROR(*val*)

This macro is used on abnormal exit from the **compile** subroutine. It should *never* contain a **return** statement. The *val* parameter is an error number. The error values and their meanings are:

Error	**Meaning**
11	Range endpoint too large.
16	Bad number.
25	\digit out of range.
36	Illegal or missing delimiter.
41	No remembered search string.
42	\\(\\) imbalance.
43	Too many \\(.
44	More than two numbers given in \\{ \\}.

45	} expected after \.
46	First number exceeds second in \{ \}.
49	[ ] imbalance.
50	Regular expression overflow.

The **compile** subroutine compiles the regular expression for later use. The *instring* parameter is never used explicitly by the **compile** subroutine, but you can use it in your macros. For instance, you may want to pass the string containing the pattern as the *instring* parameter to **compile** and use the **INIT** macro to set a pointer to the beginning of this string. (The following example uses this technique.) If your macros do not use *instring*, then call **compile** with a value of **((char *) 0)** for this parameter.

The *expbuf* parameter points to a character array where the compiled regular expression is to be placed. The *endbuf* parameter points to the location that immediately follows the character array where the compiled regular expression is to be placed. If the compiled expression cannot fit in (*endbuf-expbuf*) bytes, the call **ERROR(50)** is made.

The *eof* parameter is the character that marks the end of the regular expression. For example, in **ed** this character is usually '/' (slash).

The **regexp.h** header file defines other subroutines that perform actual regular-expression pattern matching. One of these is the **step** subroutine.

The *string* parameter of **step** is a pointer to a null-terminated string of characters to be checked for a match.

The *expbuf* parameter points to the compiled regular expression, which was obtained by a call to the **compile** subroutine.

The **step** subroutine returns the value 1 if the given string matches the pattern, and 0 if it does not match. If it matches, then **step** also sets two global character pointers: **loc1**, which points to the first character that matches the pattern, and **loc2**, which points to the character immediately following the last character that matches the pattern. Thus, if the regular expression matches the entire string, then **loc1** points to the first character of *string* and **loc2** points to the null character at the end of *string*.

The **step** subroutine uses the global variable **circf**, which is set by **compile** if the regular expression begins with a ^ (circumflex). If this variable is set, then **step** only tries to match the regular expression to the beginning of the string. If you compile more than one regular expression is before executing the first one, then save the value of **circf** for each compiled expression and set **circf** to that saved value before each call to **step**.

The **step** subroutine calls a subroutine named **advance** with the same parameters that it was passed. The **step** function increments through the *string* parameter and calls **advance** until **advance** returns a 1, indicating a match, or until the end of *string* is reached. To constrain *string* to the beginning of the string in all cases, call the **advance** subroutine directly instead of calling **step**.

When **advance** encounters an * (asterisk) or a \{ \} sequence in the regular expression, it advances its pointer to the string to be matched as far as possible and recursively calls itself trying to match the rest of the string to the rest of the regular expression. As long as there is no match, **advance** backs up along the string until it finds a match or reaches the point in the string that initially matched the * or \{ \}. It is sometimes desirable to stop this backing-up before the initial point in the string is reached. If the global character pointer **locs** is equal to the point in the string sometime during the backing up process, **advance** breaks out of the loop that backs up and returns 0. This is used by **ed** and **sed** for global substitutions on the whole line so that expressions like `s/y*//g` do not loop forever.

Example

The following is an example of the regular expression macros and calls from the **grep** command.

```
#define INIT        register char *sp=instring;
#define GETC()      (*sp++)
#define PEEKC()     (*sp)
#define UNGETC(c)   (--sp)
#define RETURN(c)   return;
#define ERROR(c)    regerr()

#include <regexp.h>
 . . .
compile (patstr, expbuf, &expbuf[ESIZE], '\0');
 . . .
if (step (linebuf, expbuf))
   succeed ( );
 . . .
```

Related Information

In this book: "NCcollate, NCcoluniq, NCeqvmap, _NCxcol, _NLxcol" on page 3-267 and "regcmp, regex" on page 3-318.

The **ed**, **grep**, and **sed** commands in *AIX Operating System Commands Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf

Purpose

Converts formatted input.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int scanf (*fmt* [, *ptr*, ...]**)**
char ****fmt*;**

int fscanf (*stream*, *fmt* [, *ptr*, ...]**)**
FILE ****stream*;**
char ****fmt*;**

int sscanf (*s*, *fmt* [, *ptr*, ...]**)**
char ****s*, ******fmt*;**

int NLscanf (*fmt* [, *ptr*, ...]**)**
char ****fmt*;**

int NLfscanf (*stream*, *fmt* [, *ptr*, ...]**)**
FILE ****stream*;**
char ****fmt*;**

int NLsscanf (*s*, *fmt* [, *ptr*, ...]**)**
char ****s*, ******fmt*;**

Description

The **scanf**, **fscanf**, and **sscanf** subroutines read character data, interpret it according to a format, and store the converted results into specified memory locations. The **NLscanf**, **NLfscanf**, and **NLsscanf** subroutines parallel their corresponding functions, providing conversion types to handle **NLchars** as well as **chars**. These subroutines read their input from the following sources:

scanf, **NLscanf**	Reads from standard input (**stdin**).
fscanf, **NLfscanf**	Reads from *stream*.
sscanf, **NLsscanf**	Reads from the character string *s*.

The *fmt* parameter contains conversion specifications used to interpret the input. The *ptr* parameters specify where to store the interpreted data.

The *fmt* parameter can contain the following:

- White space characters (blanks, tabs, new-lines, or form-feeds) which, except in two cases described following, reads the input up to the next nonwhite space character.

Unless there is a match in the control string, trailing white space (including a new-line character) is not read.

- Any character except % (percent), which must match the next character of the input stream.
- A conversion specification that directs the conversion of the next input field. It consists of the following:
  1. The character % (percent)
  2. An optional assignment suppression character, * (asterisk)
  3. An optional numeric maximum field width
  4. An optional character that sets the size of the receiving variable as for some flags, as follows:

     l Signed long integer rather than an **int** when preceding the **d**, **u**, **o** or **x** conversion codes. A double rather than a float, when preceding the **e**, **f** or **g** conversion codes.

     h Signed short integer (half **int**) rather than an **int** when preceding the **d**, **u**, **o** or **x** conversion codes.
  5. A conversion code.

The conversion specification is of the form:

%[*] [*width*] [*size*] *convcode*

The results from the conversion are placed in ****ptr*** unless you specify assignment suppression with *. Assignment suppression provides a way to describe an input field that is to be skipped. The input field is a string of nonwhite-space characters. It extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion code indicates how to interpret the input field. The corresponding *ptr* must usually be of a restricted type. You should not specify *ptr* for a suppressed field. You can use the following conversion codes:

% Accepts a single % input at this point; no assignment is done.

d Accepts a decimal integer; *ptr* should be an integer pointer.

u Accepts an unsigned decimal integer; *ptr* should be an unsigned integer pointer.

o Accepts an octal integer; *ptr* should be an integer pointer.

x Accepts a hexadecimal integer; *ptr* should be an integer pointer.

e, f, g Accepts a floating-point number. The next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to a **float**. The input format for floating-point numbers is a string of digits, with some optional characteristics:

- It can be a signed value.

- It can be an exponential value, containing a decimal point followed by an exponent field, which consists of an **E** or an **e** followed by an (optionally signed) integer.
- It can be one of the special values **INF**, **QNAN**, or **SNAN**, which is translated into the ANSI/IEEE value for infinity, quiet NaN, or signalling NaN, respectively.

s Accepts a string of **chars**. The *ptr* parameter should be a character pointer that points to an array of characters large enough to accept the string and ending with `'\0'`. The `'\0'` is added automatically. The input field ends with a white space character. A string of **chars** is output.

S (Used by the **NLscanf**, **NLfscanf**, and **NLsscanf** subroutines only.) Accepts an **NLchar** string. The *ptr* parameter should point to an array of characters large enough to accept the string and end with `'\0'`. The `'\0'` is added automatically. The input field ends with a white space character. A string of **NLchars** is output.

N (Used by the **NLscanf**, **NLfscanf**, and **NLsscanf** subroutines only.) Accepts an ASCII string, possibly containing extended character information in the form of escape sequences used by the **NLescstr** and **NLunescstr** subroutines. (See "display symbols" on page 5-24 for a list of these escape sequences.) The output is in the form of **NLchars**.

c A character is expected. The *ptr* parameter should be a character pointer. The normal skip over white space is suppressed. Use **%1s** to read the next nonwhite-space character. If a field width is given, *ptr* should refer to a character array; the indicated number of characters is read.

[*scanset*] Accepts as input the characters included in the ***scanset***. The scanset explicitly defines the characters that are accepted in the string data as those enclosed within square brackets. The normal skip over leading white space is suppressed. A scanset in the form of **[**^*scanset***]** is an ***exclusive scanset***: the ^ (circumflex) serves as a complement operator and the following characters in the scanset are not accepted as input. Conventions used in the construction of the *scanset* follow:

- You can represent a range of characters by the construct *first-last*. Thus you can express [0123456789] as [0-9]. The *first* parameter must be lexically less than or equal to *last*, or else the - (dash) stands for itself. The - also stands for itself whenever it is the first or the last character in the *scanset*.
- You can include the] (right bracket), as an element of the *scanset*, if it is the first character of the *scanset*. In this case it is not interpreted as the bracket that closes the scanset. If the scanset is an exclusive scanset, the] is preceded by the ^ (circumflex) to make the] an element of the scanset. The corresponding *ptr* must point to a character array large enough to hold the data field and that ends with `'\0'`. The `'\0'` is added automatically.

A **scanf** or **NLscanf** conversion ends at the end-of-file, the end of the control string, or when an input character conflicts with the control string. If it ends with an input character conflict, the character that conflicts is not read from the input stream.

Unless there is a match in the control string, trailing white space (including a new-line character) is not read.

The success of literal matches and suppressed assignments is not directly determinable.

Return Value

Each of these subroutines returns the ***display length*** of the string it outputs, which is the number of the display characters in the string, rather than the number of bytes. These subroutines return an **EOF** on the end of input and on a short count for missing or illegal data items.

The **scanf** and **NLscanf** subroutines return the number of successfully matched and assigned input items. This number can be 0 if there was an early conflict between an input character and the control string. If the input ends before the first conflict or conversion, only **EOF** is returned.

Examples

1. To read several values and assign them to variables:

   ```
   int i;
   float x;
   char name[50];

   scanf ("%d%f%s", &i, &x, name);
   ```

   with the input line:

   ```
   25 54.32E-1 thompson
   ```

   This assigns to `i` the value `25`, to `x` the value `5.432`, and to `name` the value `thompson\0`.

2. To perform simple pattern-matching while scanning the input:

   ```
   int i;
   float x;
   char name[50];

   scanf ("%2d%f%*d%[0-9]", &i, &x, name);
   ```

with the input:

```
56789 0123 56a72
```

This assigns `56` to `i`, `789.0` to `x`, skips `0123`, and places the string `56\0` in `name`. The next call to **getchar** (see "getc, fgetc, getchar, getw" on page 3-204) returns `a`.

Related Information

In this book: "atof, strtod" on page 3-8, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300, "getc, fgetc, getchar, getw" on page 3-204, "standard i/o library" on page 3-342, "strtol, atol, atoi" on page 3-347, and "display symbols" on page 5-24.

Examples of using **scanf** in *C Language Guide and Reference.*

"Overview of International Character Support" in *IBM RT PC Managing the AIX Operating System.*

setbuf, setvbuf

Purpose

Assigns buffering to a stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

void setbuf (*stream*, *buf***)**
**FILE ****stream*;
**char ****buf*;

int setvbuf (*stream*, *buf*, *type*, *size***)**
**FILE ****stream*;
**char ****buf*;
int *type*, *size*;

Description

The **setbuf** subroutine causes the character array pointed to by the *buf* parameter to be used instead of an automatically allocated buffer. Use the **setbuf** subroutine after a stream has been opened but before it is read or written.

If the *buf* parameter is a **NULL** character pointer, input/output is completely unbuffered.

A constant, **BUFSIZ**, defined in the **stdio.h** header file, tells how big an array is needed:

```
char buf[BUFSIZ];
```

For the **setvbuf** subroutine, *type* determines how *stream* is buffered:

_IOFBF Causes input/output to be fully buffered.

_IOLBF Causes output to be line buffered. The buffer is flushed when a new line is written, the buffer is full, or input is requested.

_IONBUF Causes input/output to be completely unbuffered.

If the *buf* parameter is not a **NULL** character pointer, the array it points to is used for buffering instead of an automatically allocated buffer. *size* specifies the size of the buffer to be used. The constant **BUFSIZ** in **stdio.h** can be a good buffer size. If input/output is unbuffered, *buf* and *size* are ignored.

A buffer is normally obtained from the **malloc** subroutine at the time of the first **getc** or **putc** on the file, except that the standard error stream, **stderr**, is normally not buffered.

Output streams directed to terminals are always either line-buffered or unbuffered.

Note: A common source of error is allocating buffer space as an automatic variable in a code block, and then failing to close the stream in the same block.

Related Information

In this book: "fopen, freopen, fdopen" on page 3-168, "getc, fgetc, getchar, getw" on page 3-204, "malloc, free, realloc, calloc" on page 3-236, "putc, putchar, fputc, putw" on page 3-309, and "standard i/o library" on page 3-342.

setjmp, longjmp

Purpose

Saves and restores the current execution context.

Library

Standard C Library (**libc.a**)

Syntax

#include <setjmp.h>

int setjmp (*ctxt***)**
jmp_buf *ctxt*;

void longjmp (*ctxt*, *val***)**
jmp_buf *ctxt*;
int *val*;

Description

The **setjmp** and **longjmp** subroutines can be useful when handling errors and interrupts encountered in low-level subroutines of a program.

The **setjmp** subroutine saves the current stack context and signal mask in the buffer specified by the *ctxt* parameter. The **setjmp** subroutine returns a value of 0.

The **longjmp** subroutine restores the stack context and signal mask that were saved by the **setjmp** subroutine in the corresponding *ctxt* buffer. After the **longjmp** subroutine has completed, the program execution continues as if the corresponding call to **setjmp** had just returned the value of the *val* parameter. The subroutine that called **setjmp** must not have returned before the completion of the **longjmp** subroutine.

The **longjmp** subroutine cannot return 0 to the previous context. The value 0 is reserved to indicate the actual return from the **setjmp** subroutine when first called by the program. If the **longjmp** subroutine is passed a *val* parameter of 0, then execution continues as if the corresponding call to the **setjmp** subroutine had returned a value of 1. All accessible data have values as of the time the **longjmp** subroutine is called.

Warning: If the **longjmp** subroutine is called with a *ctxt* parameter that was not previously set by **setjmp**, or if the subroutine that made the corresponding call to **setjmp** has already returned, then the results of the **longjmp** subroutine are undefined.

Related Information

In this book: "signal" on page 2-145 and "sigvec" on page 2-156.

sgetl, sputl

Purpose

Accesses long numeric data in a machine-independent fashion.

Library

Object File Access Routine Library (**libld.a**)

Syntax

long sgetl (*buffer***)**
char ****buffer*;**

void sputl (*value***,** *buffer***)**
long *value***;**
char ****buffer*;**

Description

The **sgetl** subroutine retrieves 4 bytes from memory starting at the location pointed to by the **buffer** parameter. It then returns the bytes as a **long** value with the byte ordering of the host machine.

The **sputl** subroutine stores the 4 bytes of the *value* parameter into memory starting at the location pointed to by the *buffer* parameter. The order of the bytes is the same across all machines.

Using **sputl** and **sgetl** subroutines together provides a machine-independent way of storing long numeric data in an ASCII file. For example, the numeric data stored in the portable archive file format is accessed with the **sputl** and **sgetl** subroutines.

Related Information

In this book: "frexp, ldexp, modf" on page 3-194 and "ar" on page 4-18.

sin, cos, tan, asin, acos, atan, atan2

Purpose

Computes trigonometric functions.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double sin (*x*)
double *x*;

double cos (*x*)
double *x*;

double tan (*x*)
double *x*;

double asin (*x*)
double *x*;

double acos (*x*)
double *x*;

double atan (*x*)
double *x*;

double atan2 (*y*, *x*)
double *x*, *y*;

Description

The **sin**, **cos**, and **tan** subroutines return the sine, cosine and tangent, respectively, of their parameters, which are in radians.

The **asin** subroutine returns the arcsine of *x*, in the range -π/2 to π/2.

The **acos** subroutine returns the arccosine of *x*, in the range 0 to π.

The **atan** subroutine returns the arctangent of *x*, in the range -π/2 to π/2.

The **atan2** subroutine returns the arctangent of *y*/*x*, in the range -π to π, using the signs of both parameters to determine the quadrant of the return value.

Diagnostics

These subroutines can perform either of the following types of error handling. Both types of error handling allow you to define special actions to be taken when an error occurs.

1. By default, **matherr** error handling is performed, as described on page 3-238. The default error-handling procedures for these subroutines are as follows:

   sin, cos, tan
   The **sin**, **cos** and **tan** subroutines lose accuracy when passed a large value for the *x* parameter. For sufficiently large parameters, these functions return 0 when there would otherwise be a complete loss of significance. In this case, a message that indicates a **TLOSS** error is written to standard error. For less extreme values, a **PLOSS** error is generated but no message is written. In both cases, **errno** is set to **ERANGE**.

   The **tan** subroutine returns **HUGE** if its parameter is near an odd multiple of $\pi/2$ when the correct value would overflow, and sets **errno** to **ERANGE**.

   asin, acos
   The **asin** and **acos** subroutines return 0 and set **errno** to **EDOM** if their parameters are larger than 1.0. In addition, an error message that indicates a domain error is written to the standard error output.

2. Exception handling can also be performed according to ANSI/IEEE standard 754-1985 for binary floating-point arithmetic, as discussed under "Exception Handling" on page 3-186. To select ANSI/IEEE exception handling, define the **_C_func** preprocessor variable. You can do this by inserting the statement **#define _C_func** before the **#include <math.h>**, or by specifying the **-D_C_func** flag to the **cc** command when compiling the program.

   If a hardware floating-point processor is installed in your system, then using this option can provide greater performance in addition to IEEE exception handling. Defining **_C_func** causes the **math.h** header file to define macros that make the names **sin, cos, tan**, . . . appear to the compiler as **_C_sin**, **_C_cos**, **_C_tan**, ... These special names instruct the C compiler to generate code that avoids the overhead of the math library subroutines and issues compatible-mode floating-point calls directly. See "fpfp" on page 3-170 for information about compatible mode.

Related Information

In this book: "fpfp" on page 3-170, "Exception Handling" on page 3-186, and "matherr" on page 3-238.

sinh, cosh, tanh

Purpose

Computes hyperbolic functions.

Library

Math Library (**libm.a**)

Syntax

#include <math.h>

double sinh (*x*)
double *x*;

double cosh (*x*)
double *x*;

double tanh (*x*)
double *x*;

Description

The **sinh** subroutine returns the hyperbolic sine of its parameter. The **cosh** subroutine returns the hyperbolic cosine of its parameter. The **tanh** subroutine returns the hyperbolic tangent of its parameter.

The **sinh** and the **cosh** subroutines return **HUGE** if the correct value overflows. **errno** is also set to **ERANGE**.

You can use the **matherr** subroutine to change these error-handling procedures. See "matherr" on page 3-238 for details.

sleep

Purpose

Suspends execution of the current process for an interval of time.

Library

Standard C Library (**libc.a**)

Syntax

unsigned int sleep (*seconds***)**
unsigned int *seconds***;**

Description

The **sleep** subroutine causes the current process to suspend execution for the number of seconds specified by the *seconds* parameter. The sleep routine sets an alarm and pauses until that alarm or some other signal occurs.

The actual sleep time of the process may be either shorter or longer than the requested sleep time. The sleep time may be shorter because:

- Wakeups occur on the second at fixed 1-second intervals according to an internal clock.
- Any caught signal terminates the sleep following execution of that signal's catching routine.

The sleep time may be longer than the requested sleep time due to the scheduling of other activities in the system.

The value returned by the **sleep** subroutine is the requested sleep time minus the time actually slept.

The process calling the **sleep** subroutine may have set an alarm prior to calling the **sleep** subroutine.

If a previous alarm has been set, and the **sleep** subroutine's sleep time exceeds the process's previously set sleep time, the process only sleeps until the time specified by the previously set alarm and the calling process's alarm catch routine is executed just before the **sleep** subroutine returns.

If a previous alarm has been set, and the **sleep** subroutine's sleep time is less than the process's previously set sleep time, the current process is suspended from execution for the number of seconds specified by the **sleep** subroutine. The previously set alarm is reset to go off at the same time it would have without the **sleep** subroutines intervention.

Warning: The results are undefined if, while it is sleeping, the calling program issues any other **alarm** or **sleep** calls. This can happen if a signal arrives in the interim and the signal handler calls **alarm** or **sleep**.

Related Information

In this book: "alarm" on page 2-13, "pause" on page 2-94, and "signal" on page 2-145.

ssignal, gsignal

Purpose

Implements a software signal facility.

Library

Standard C Library (**libc.a**)

Syntax

#include <signal.h>

int (*ssignal (*sig*, *action***)) ( )**
int *sig*, **(****action***) ( );**

int gsignal (*sig***)**
int *sig***;**

Description

The **ssignal** and **gsignal** subroutines implement a software facility similar to that of the **signal** and **kill** system calls. However, there is no connection between the two facilities. User programs can use **ssignal** and **gsignal** to handle exceptional processing within an application. **signal** and related system calls handle system-defined exceptions.

The software signals available are associated with integers in the range 1 through 15. Other values are reserved for use by the C library and should not be used.

The **ssignal** subroutine associates the procedure specified by the *action* parameter with the software signal specified by the *sig* parameter. The **gsignal** subroutine "raises" the signal *sig*, causing the procedure specified by the *action* parameter to be taken.

The *action* parameter is either a pointer to a user-defined subroutine, or one of the constants **SIG_DFL** (default action) and **SIG_IGN** (ignore signal). The **ssignal** subroutine returns the procedure that was previously established for that signal. If no procedure was established before, or if the signal number is illegal, then **ssignal** returns the value **SIG_DFL**.

The **gsignal** subroutine "raises" the signal specified by the *sig* parameter by doing the following:

- If the procedure for *sig* is **SIG_DFL**, then the **gsignal** subroutine returns a value of 0 and takes no other action.
- If the procedure for *sig* is **SIG_IGN**, then the **gsignal** subroutine returns a value of 1 and takes no other action.
- If the procedure for *sig* is a subroutine, then the *action* value is reset to **SIG_DFL** and the subroutine is called with *sig* passed as its parameter. The **gsignal** subroutine returns the value that is returned by the signal-handling subroutine.
- If the procedure for *sig* is an illegal value or if no procedure was ever specified for that signal, then **gsignal** returns a value of 0 and takes no other action.

Related Information

In this book: "kill" on page 2-60 and "signal" on page 2-145.

standard i/o library

Purpose

Performs standard buffered input and output operations.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

FILE *stdin, *stdout, *stderr;

Description

These macros and subroutines provide an efficient user-level I/O buffering scheme.

The in-line macros **getc** and **putc** handle characters quickly. The following macros and subroutines all use the **getc** and **putc** macros:

getchar macro
putchar macro
fgetc subroutine
fgets subroutine
fprintf subroutine
fputc subroutine
fputs subroutine
fread subroutine
fscanf subroutine
fwrite subroutine
gets subroutine
getw subroutine
printf subroutine
puts subroutine
putw subroutine
scanf subroutine

A file with associated buffering is called a ***stream*** and is declared to be a pointer to the defined type **FILE**. The **fopen** subroutine constructs descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the **stdio.h** header file and associated with the standard open streams:

stdin	Standard input stream
stdout	Standard output stream
stderr	Standard error output stream.

The constant **NULL** (0) designates a special pointer value that does not point to any data structure.

Most integer subroutines that deal with streams return the constant **EOF** (-1) upon end-of-file or an error. See each individual subroutine for detailed information about the return value.

Programs that use this input/output package must include the header file of pertinent macro definitions, as follows:

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

The subroutines and constants in the input/output package are declared in the header file and do not need any further declaration. The constants and the following routines are implemented as macros. Redeclaration of these names is not allowed.

getc
getchar
putc
putchar

feof
ferror
clearerr
fileno

Warning: Invalid stream pointers usually cause errors, possibly including program termination. Individual subroutine descriptions describe the possible error conditions.

Related Information

In this book: "close" on page 2-25, "lseek" on page 2-67, "open" on page 2-90, "pipe" on page 2-95, "read, readx" on page 2-106, "write, writex" on page 2-184, "ctermid" on page 3-44, "cuserid" on page 3-62, "fclose, fflush" on page 3-163, "feof, ferror, clearerr, fileno" on page 3-165, "fopen, freopen, fdopen" on page 3-168, "fread, fwrite" on page 3-192, "fseek, rewind, ftell" on page 3-196, "getc, fgetc, getchar, getw" on page 3-204, "gets, fgets" on page 3-221, "popen, pclose" on page 3-298, "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300, "putc, putchar, fputc, putw" on page 3-309, "puts, fputs" on page 3-313, "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325, "setbuf" on page 3-330, "system" on page 3-350, "tmpfile" on page 3-354, "tmpnam, tempnam" on page 3-355, and "ungetc" on page 3-369.

string

Purpose

Performs operations on strings.

Library

Standard C Library (**libc.a**)

Syntax

```
#include <string.h>

char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

Description

The **string** subroutines copy, compare, and append strings in memory, and they determine such things as location, size, and existence of strings in memory.

The parameters *s1*, *s2* and *s* point to strings. A string is an array of characters terminated by a null character. The subroutines **strcat**, **strncat**, **strcpy**, and **strncpy** all alter *s1*. They do not check for overflow of the array pointed to by *s1*. All string movement is performed character by character and starts at the left. Overlapping moves toward the left work as expected, but overlapping moves to the right may give unexpected results. All of these subroutines are declared in the **string.h** header file.

The **strcat** subroutine adds a copy of the string pointed to by the *s2* parameter to the end of the string pointed to by the *s1* parameter. The **strcat** subroutine returns a pointer to the null-terminated result.

The **strncat** subroutine copies at most *n* bytes of *s2* to the end of the string pointed to by the *s1* parameter. Copying stops before *n* bytes if a null character is encountered in the *s2* string. The **strncat** subroutine returns a pointer to the null-terminated result.

The **strcmp** subroutine lexicographically compares the string pointed to by the *s1* parameter to the string pointed to by the *s2* parameter. The **strcmp** subroutine uses native character comparison, which may be signed or unsigned. The **strcmp** subroutine returns a value that is:

Less than 0	If *s1* is less than *s2*
Equal to 0	If *s1* is equal to *s2*
Greater than 0	If *s1* is greater than *s2*.

The **strncmp** subroutine makes the same comparison as **strcmp**, but it compares at most *n* pairs of characters.

The **strcpy** subroutine copies the string pointed to by the *s2* parameter to the character array pointed to by the *s1* parameter. Copying stops when the null character is copied. The **strcpy** subroutine returns the value of the *s1* parameter.

The **strncpy** subroutine copies *n* bytes from the string pointed to by the *s2* parameter to the character array pointed to by the *s1* parameter. If *s2* is less than *n* characters long, then **strncpy** pads *s1* with trailing null characters to fill *n* bytes. If *s2* is *n* or more characters long, then only the first *n* characters are copied and the result is not terminated with a null character. The **strncpy** subroutine returns the value of the *s1* parameter.

The **strlen** subroutine returns the number of characters in the string pointed to by thes parameter, not including the terminating null character.

The **strchr** subroutine returns a pointer to the first occurrence of the character specified by the *c* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the character does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **strrchr** subroutine returns a pointer to the last occurrence of the character specified by the *c* parameter in the string pointed to by the *s* parameter. A **NULL** pointer is returned if the character does not occur in the string. The null character that terminates a string is considered to be part of the string.

The **strpbrk** subroutine returns a pointer to the first occurrence in the string pointed to by the *s1* parameter of any character from the string pointed to by the *s2* parameter. A **NULL** pointer is returned if no character matches.

The **strspn** subroutine returns the length of the initial segment of the string pointed to by the *s1* parameter that consists entirely of characters from the string pointed to by the *s2* parameter.

The **strcspn** subroutine returns the length of the initial segment of the string pointed to by the *s1* parameter that consists entirely of characters *not* from the string pointed to by the *s2* parameter.

The **strtok** subroutine returns a pointer to an occurrence of a text token in the string pointed to by the *s1* parameter. The *s2* parameter specifies a set of token delimiters. If the *s1* parameter is anything other than **NULL**, then the **strtok** subroutine reads the string pointed to by the *s1* parameter until it finds one of the delimiter characters specified by the *s2* parameter. It then stores a null character into the string, replacing the delimiter, and returns a pointer to the first character of the text token. The **strtok** subroutine keeps track of its position in the string so that subsequent calls with a **NULL** *s1* parameter step through the string. The delimiters specified by the *s2* parameter can be changed for subsequent calls to **strtok**. When no tokens remain in the string pointed to by the *s1* parameter, the **strtok** subroutine returns a **NULL** pointer.

Related Information

In this book: "memccpy, memchr, memcmp, memcpy, memset" on page 3-245, "NCstring" on page 3-272, "NLstring" on page 3-285, and "swab" on page 3-349.

strtol, atol, atoi

Purpose

Converts a string to an integer.

Library

Standard C Library (**libc.a**)

Syntax

long strtol (*str*, *ptr*, *base***)**
**char ****str*, ***ptr*;
int *base*;

long atol (*str***)**
**char ****str*;

int atoi (*str***)**
**char ****str*;

Description

The **strtol** subroutine returns a long integer whose value is represented by the character string *str*. **strtol** scans the string up to the first character that is inconsistent with the *base*. Leading white-space characters are ignored.

Warning: Overflow conditions are ignored.

If the value of *ptr* is not **(char **) NULL**, then a pointer to the character that terminated the scan is stored in **ptr*. If an integer cannot be formed, **ptr* is set to *str*, and 0 is returned.

If the *base* parameter is positive and not greater than 36, then it is used as the base for conversion. After an optional leading sign, leading zeroes are ignored. `0x` or `0X` is ignored if *base* is 16.

If the *base* parameter is 0, the string determines the base. Thus, after an optional leading sign, a leading 0 indicates octal conversion, and a leading `0x` or `0X` indicates hexadecimal conversion. The default is to use decimal conversion.

Note: Truncation from **long** to **int** can take place upon assignment, or by an explicit cast.

The **atol** (*str*) subroutine call is equivalent to **strtol** (*str*, **(char **) NULL, 10)**.

The **atoi** (*str*) subroutine call is equivalent to **(int) strtol** (*str*, **(char **) NULL, 10)**.

The **atoi** and **atol** subroutines do not actually call **strtol**.

The **strtol**, **atol**, and **atoi** subroutines perform conversions to integers. See "atof, strtod" on page 3-8 for information on conversions to floating-point numbers.

Related Information

In this book: "atof, strtod" on page 3-8 and "scanf, fscanf, sscanf, NLscanf, NLfscanf, NLsscanf" on page 3-325.

swab

Purpose

Copies bytes.

Library

Standard C Library (**libc.a**)

Syntax

void swab (*from*, *to*, *nbytes*)
char **from*, **to*;
int *nbytes*;

Description

The **swab** subroutine copies *nbytes* bytes from the location pointed to by the *from* parameter to the array pointed to by the *to* parameter, exchanging adjacent even and odd bytes.

The *nbytes* parameter should be even and nonnegative. If the *nbytes* parameter is odd and positive, the **swab** uses *nbytes*-1 instead. If the *nbytes* parameter is negative, then **swab** does nothing.

Related Information

In this book: "memccpy, memchr, memcmp, memcpy, memset" on page 3-245 and "string" on page 3-344.

system

Purpose

Runs a shell command.

Library

Standard C Library (**libc.a**)

Syntax

#include <stdio.h>

int system (*string*)
char ****string*;**

Description

The **system** subroutine passes the *string* parameter to the **sh** command as input. Then **sh** interprets *string* as a command and runs it.

The **system** subroutine invokes the **fork** system call to create a child process that in turn uses **exec** to run **/bin/sh**, which interprets the shell command contained in the *string* parameter. The current process waits until the shell has completed, then returns the exit status of the shell.

Note: The **system** subroutine runs only **sh** shell commands (also called ***Bourne shell*** commands). The results are unpredictable if the *string* parameter is not a valid **sh** shell command.

Return Value

Upon successful completion, the **system** subroutine returns the exit status of the shell. See “wait” on page 2-182 for an explanation of the exit status.

If the **fork** fails, then **system** returns a value of -1. If the **exec** fails, then it returns 127. It either case, **errno** is set to indicate the error.

File

/bin/sh

Related Information

In this book: "exec: execl, execv, execle, execve, execlp, execvp" on page 2-34, "exit, _exit" on page 2-40, "fork" on page 2-46, and "wait" on page 2-182.

The **sh** command in *AIX Operating System Commands Reference.*

termdef

Purpose

Queries terminal characteristics.

Library

Standard C Library (**libc.a**)

Syntax

char *termdef (*fildes***,** *c***)**
int *fildes***;**
char *c***;**

Description

The **termdef** subroutine returns a pointer to a null-terminated static character string that identifies a characteristic of the terminal that is open on the file descriptor specified by the *fildes* parameter. The *c* parameter specifies the characteristic that is to be queried. **termdef** determines this information by performing the following actions:

1. It queries the terminal device, using the Query HFT Device command, which is discussed on page 6-47.
2. If the query fails, then **termdef** uses the value of an environment variable.
3. If the environment variable is not set, then **termdef** returns the default value specified in the following table.

The following list shows the valid request types and the corresponding environment variables that are used if the Query HFT Device command fails:

	Environment	**Default**	
c	**Variable**	**Value**	**Description**
t	**TERM**	`"ibm5151"`	The terminal type
l	**LINES**	**NULL**	The number of lines or rows, based on the current font

c	**Environment Variable**	**Default Value**	**Description**
c	**COLUMNS**	**NULL**	The number of character columns, based on the current font.

Note: When *fildes* identifies an asynchronous terminal, the Query HFT Device command always fails and the environment variable is always checked. The **TERM** variable is automatically set each time you log in. **LINES** and **COLUMNS** need to be set only if:

- You are using an asynchronous terminal and want to override the **lines** and **cols** settings in the **terminfo** data base, or
- Your asynchronous terminal has an unusual number of lines or columns and you are running an application that uses **termdef**, but not **terminfo**.

This is true because the **terminfo** initialization subroutine, **setupterm**, calls **termdef** to determine the number of lines and columns on the display. If **termdef** cannot supply this information, then **setupterm** uses the values in the **terminfo** data base.

Related Information

In this book: "Terminfo Level Subroutines" on page 3-57, "terminfo" on page 4-148, and "Query HFT Device Command" on page 6-47.

The **display** and **termdef** commands in *AIX Operating System Commands Reference.*

tmpfile

Purpose

Creates a temporary file.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

FILE *tmpfile ( )

Description

The **tmpfile** subroutine creates a temporary file and returns its FILE pointer. The file is opened for update. If Distributed Services is installed on your system, this file can reside on a remote node. The temporary file is automatically deleted when the process using it terminates.

If the file cannot be opened, **tmpfile** writes an error message to the standard error output and returns a **NULL** pointer.

Related Information

In this book: "creat" on page 2-27, "unlink" on page 2-174, "fopen, freopen, fdopen" on page 3-168, "mktemp" on page 3-247, "standard i/o library" on page 3-342, and "tmpnam, tempnam" on page 3-355.

tmpnam, tempnam

Purpose

Constructs the name for a temporary file.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

char *tmpnam (*s***)**
char ****s*;**

char *tempnam (*dir*, *pfx***)**
char ****dir*, ******pfx*;**

Description

The **tmpnam** and **tempnam** subroutines generate file names for temporary files.

The **tmpnam** subroutine generates a file name using the path name defined as **P_tmpdir** in the **stdio.h** header file. If the *s* parameter is **NULL**, the **tmpnam** subroutine places its result into an internal static area and returns a pointer to that area. The next call to this subroutine destroys the contents of the area.

If the *s* parameter is not **NULL**, it is assumed to be the address of an array of at least the number of bytes specified by **L_tmpnam**. **L_tmpnam** is a constant defined in **stdio.h**. The **tmpnam** subroutine places its results into that array and returns the value of the *s* parameter.

The **tempnam** subroutine allows you to control the choice of a directory. The *dir* parameter points to the path name of the directory in which the file is to be created. If the *dir* parameter is **NULL** or points to a string which is not a path name for an appropriate directory, the path name defined as **P_tmpdir** in the **stdio.h** header file is used. If that path name is not accessible, **/tmp** is used. You can bypass the selection of a path name by providing an environment variable, **TMPDIR**, in the user's environment. The value of the **TMPDIR** variable is a path name for the desired temporary-file directory. If the **TMPDIR** variable is used, both the *dir* parameter and **L_tmpnam** are ignored.

The *pfx* parameter of the **tempname** subroutine allows you to specify an initial character sequence with which the file name begins. The *pfx* parameter can be **NULL**, or it can point to a string of up to five characters to be used as the first few characters of the temporary file name.

The **tempnam** subroutine uses the **malloc** subroutine to obtain space for the constructed file name. The return value is a pointer to this space. Therefore, the pointer value returned by **tempnam** can be used as a parameter to the **free** subroutine.

If the **tempnam** subroutine cannot return the expected result for any reason (for example, if the **malloc** subroutine fails, or if an appropriate directory cannot be found), then it returns a **NULL** pointer.

Warning: The **tmpnam** and **tempnam** subroutines generate a different file name each time they are called. If they are called more than 4,096 times by a single process, they start recycling previously used names.

Files created using these subroutines reside in a directory intended for temporary use, and their names are unique. It is your responsibility to use the **unlink** system call to remove the file when no longer needed.

Between the time a file name is created and the file is opened, it is possible for some other process to create a file with the same name. This should not happen if that other process uses these subroutines or the **mktemp** subroutine, and if the file names are chosen to make duplication by other means unlikely.

Related Information

In this book: "creat" on page 2-27, "unlink" on page 2-174, "fopen, freopen, fdopen" on page 3-168, "malloc, free, realloc, calloc" on page 3-236, "mktemp" on page 3-247, "tmpfile" on page 3-354, and "environment" on page 5-47.

trace_on

Purpose

Checks whether trace channel is enabled.

Library

Run-time Services Library (**librts.a**)

Syntax

int trace_on (*chanmask***)**
unsigned long *chanmask***;**

Description

The **trace_on** subroutine queries the application trace device driver to determine whether a given trace channel is enabled. **trace_on** allows a program to avoid the unnecessary overhead of setting up the trace message when its trace channel is disabled. **trace_on** is a C run-time subroutine and should be used by application programs, but not by device drivers.

The *chanmask* parameter is a mask with the bit corresponding to the channel number set. It can be formed by the expression `(1 << 31 -` *channum*`)`. User programs can use only channel number 31, which means that the value of *chanmask* must be 1 for user programs.

Making repeated calls to the trace device driver involves significant overhead, so call **trace_on** only once: either at the start of processing or just before the first trace point in the program.

If the application trace device driver is not already open, **trace_on** opens it.

Upon successful completion, **trace_on** returns 1 if the channel is enabled, or 0 if the channel is disabled. If the **trace_on** subroutine fails, a message is written to the standard error output, and a value of -1 is returned.

File

/dev/appltrace

Related Information

In this book: "trcunix" on page 3-362, "trace" on page 6-128, and "Trace Logging" on page C-32.

The **trace** command in *AIX Operating System Commands Reference.*

The discussion of trace in *AIX Operating System Programming Tools and Interfaces.*

trc_start, trc_stop

Purpose

Starts and stops a trace daemon.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <sys/trace.h>

trc_start (*outpath*, *entsize*, *numents*, *lastonly*, *trcinfo***)**
**char ****outpath*;
int *entsize*, *numents*, *lastonly*;
**struct t_struct ****trcinfo*;

trc_stop (*pid***)**
int *pid*;

Description

The **trc_start** subroutine starts a trace daemon process when called from within a running application program. This subroutine can be used to start an ordinary (***monitor***) trace session or a VRM-specific ***generic*** trace session. In a generic trace session, as many as seven daemons can trace different devices simultaneously. In a monitor trace session, only one daemon can run at any one time since monitor trace classes are fixed. For information about generic and monitor traces, see the **trace** command in *AIX Operating System Commands Reference.*

The *outpath* parameter is a pointer to the path name of the trace output file. For generic traces, the data from each device can be saved in a separate file by specifying a unique file name when starting each daemon. If the *outpath* parameter is a **NULL** pointer, then the default file name is used. The default file name is the value of the **file** keyword in the **/dev/trace** stanza of the **/etc/rasconf** file.

The *entsize* parameter indicates the size in bytes of each trace entry that the application intends to send to the VRM trace collector. If *entsize* is 0, then the default entry size, 40 bytes, is used.

Warning: You must specify a file name in the *outpath* parameter if the value of *entsize* is not 0, that is, if you specify a trace entry size other than the default. Failure to give a file name in this case results in unusable data if another trace daemon uses the default file name.

The *numents* parameter indicates the maximum number of entries to keep in the kernel's trace buffer. If *numents* is 0, then the default value is used. The default number of entries is the value of the **buffer** keyword in the **/dev/trace** stanza of the **/etc/rasconf** file.

The *lastonly* parameter can have a value of 1, indicating that only the last buffer of data is to be saved in the trace file, or a value of 0, instructing the daemon to save all data continuously.

The *trcinfo* parameter is a pointer to a **t_struct** structure into which the **trc_start** subroutine stores information about the trace daemon that it starts. This structure is defined in the **sys/trace.h** header file, and it contains the following members:

t_PID The process ID of the daemon.

buff_addr The address of the trace buffer allocated in the AIX kernel for use in a generic trace session.

buff_len The size in bytes of the trace buffer allocated in the AIX kernel.

channel_ID The bit mask to use in the *traceid* field when recording trace data. This value is the channel number shifted left 11 bits with a hook ID of 0, as described in "trcunix" on page 3-362.

The **trc_stop** subroutine **kill**s a trace daemon process that was started by **trc_start**. Its only parameter is the process ID of the trace daemon, which **trc_start** provides by storing it in *trcinfo*->**t_PID**.

Return Value

The **trc_start** subroutine returns a value of 0 upon successful completion. If unsuccessful, then **trc_start** returns the value of **errno** that was set by the failing system call, and an error message is written to the standard error output. **trc_start** can fail when it invokes the **pipe**, **fork**, **exec**, or **read** system call.

The **trc_stop** subroutine returns 0 if successful, or -1 if the **kill** system call fails.

Related Information

In this book: "trace_on" on page 3-357, "trcunix" on page 3-362, "rasconf" on page 4-133, and "trace" on page 6-128.

The **trace** command in *AIX Operating System Commands Reference.*

The discussion about using the trace subroutines in *AIX Operating System Programming Tools and Interfaces.*

trcunix

Purpose

Records application trace log entries.

Library

Run-time Services Library (**librts.a**)

Syntax

int trcunix (*buf*, *cnt***)**
char ****buf*;**
unsigned int *cnt***;**

Description

The **trcunix** subroutine invokes the application trace device driver to record a trace log entry. **trcunix** is a C run-time subroutine. Device drivers should use the **trsave** subroutine to log trace events.

The *buf* parameter points to a buffer containing a 2-byte *traceid* followed by up to 20 bytes of user-defined trace data. The high-order 5 bits of the *traceid* specify the channel number, and the low-order 11 bits specify the hook ID for the message. User programs may use only channel number 31. The *cnt* parameter specifies the number of bytes in the buffer, including the *traceid.*

If the application trace device driver is not open, then **trcunix** opens it before writing the trace log entry to it.

Return Value

Upon successful completion, a value of 0 is returned and a trace log entry is written to **/dev/appltrace**. If the **trcunix** subroutine fails, an error message is written to the standard error output, and a value of -1 is returned.

File

/dev/appltrace

Related Information

In this book: "trace_on" on page 3-357, "trace" on page 6-128, and "Trace Logging" on page C-32.

The **trace** command in *AIX Operating System Commands Reference.*

The discussion of trace in *AIX Operating System Programming Tools and Interfaces.*

tsearch, tdelete, twalk

Purpose

Manages binary search trees.

Library

Standard C Library (**libc.a**)

Syntax

#include <search.h>

char *tsearch ((char *) *key*, **(char **)** *rootp*, *compar*)
int (***compar*) **( );**

char *tdelete ((char *) *key*, **(char **)** *rootp*, *compar*)
int (***compar*) **( );**

void twalk ((char *) *root*, *action*)
void (***action*) **( );**

Description

The **tsearch** subroutine performs a binary tree search. The algorithm is generalized from Donald E. Knuth's *The Art of Computer Programming*, Volume 3, 6.2.2, Algorithm T.* It returns a pointer into a tree indicating where the data specified by the *key* parameter can be found. If the data specified by the *key* parameter is not found, the data is added to the tree in the correct place. If there is not enough space available to create a new node, a NULL pointer is returned. The *rootp* parameter points to a variable that points to the root of the tree. If the *rootp* parameter is **NULL**, the variable is set to point to the root of a new tree.

* Reading, Massachusetts: Addison-Wesley, 1981.

The *compar* parameter is a pointer to the comparison function, which is called with two parameters that point to the elements being compared. The comparison function must compare its parameters and return a value as follows:

- If the first parameter is less than the second parameter, *compar* must return a value less than 0.
- If the first parameter is equal to the second parameter, *compar* must return 0.
- If the first parameter is greater than the second parameter, *compar* must return a value greater than 0.

The comparison function need not compare every byte, so arbitrary data can be contained in the elements in addition to the values being compared.

If the *rootp* parameter is **NULL** on entry, then a **NULL** pointer is returned.

The **tdelete** subroutine deletes the data specified by the *key* parameter. It is generalized from Knuth (6.2.2) Algorithm D. The *rootp* and *compar* parameters perform the same function as they do for the **tsearch** subroutine. The variable pointed to by the *rootp* parameter will be changed if the deleted node is the root of the binary tree. The **tdelete** subroutine returns a pointer to the parent node of the deleted node. If the data is not found, a **NULL** pointer is returned. If the *rootp* parameter is **NULL** on entry, then a **NULL** pointer is returned.

The **twalk** subroutine steps through the binary search tree whose root is pointed to by the *root* parameter. (Any node in a tree can be used as the root to step through the tree below that node.) The *action* parameter is the name of a routine to be invoked at each node. The routine specified by the *action* parameter is called with three parameters. The first parameter is the address of the node currently being pointed to. The second parameter is a value from an enumeration data type

```
    typedef enum {preorder, postorder, endorder, leaf}  VISIT;
```

(This data type is defined in the **search.h** header file). The actual value of the second parameter depends on whether this is the first, second, or third time that the node has been visited during a depth-first, left-to-right traversal of the tree, or whether the node is a ***leaf***. A leaf is a node that is not the parent of another node. The third parameter is the level of the node in the tree, with the root node being level zero.

The pointers to the key and the root of the tree should be of type pointer-to-element and cast to type pointer-to-character. Although declared as type pointer-to-character, the value returned should be cast into type pointer-to-element.

Related Information

In this book: "bsearch" on page 3-11, "hsearch, hcreate, hdestroy" on page 3-227, and "lsearch" on page 3-234.

ttyname, isatty

Purpose

Gets the name of a terminal.

Library

Standard C Library (**libc.a**)

Syntax

char *ttyname (*fildes***)**
int *fildes*;

int isatty (*fildes***)**
int *fildes*;

Description

The **ttyname** subroutine gets the name of a terminal. It returns a pointer to a string containing the null-terminated path name of the terminal device associated with file descriptor specified by the *fildes* parameter. A **NULL** pointer is returned if the file descriptor does not describe a terminal device in directory **/dev**.

The **isatty** subroutine determines if the device associated with the file descriptor specified by the *fildes* parameter is a terminal. If the specified file descriptor is associated with a terminal, the **isatty** subroutine returns a value of 1. If the file descriptor is not associated with a terminal, a value of 0 is returned.

The return value of **ttyname** points to static data whose contents are overwritten by each call.

Files

/dev/*

ttyslot

Purpose

Finds the slot in the **utmp** file for the current user.

Library

Standard C Library (**libc.a**)

Syntax

int ttyslot ( )

Description

The **ttyslot** subroutine returns the index of the current user's entry in the **/etc/utmp** file. The **ttyslot** subroutine scans the **/etc/utmp** file for the name of the terminal associated with the standard input, the standard output, or the error output (0, 1, or 2).

The **ttyslot** subroutine returns 0 if an error was encountered while searching for the terminal name, or if none of the first three file descriptors (0, 1, and 2) is is associated with a terminal device.

Files

/etc/inittab
/etc/utmp

Related Information

In this book: "getutent" on page 3-224 and "ttyname, isatty" on page 3-367.

ungetc

Purpose

Pushes a character back into input stream.

Library

Standard I/O Package (**libc.a**)

Syntax

#include <stdio.h>

int ungetc (*c*, *stream***)**
int *c***;**
FILE **stream***;**

Description

The **ungetc** subroutine inserts the character specified by the *c* parameter into the buffer associated with the input stream specified by the *stream* parameter. This causes the next call to the **getc** subroutine to return *c*. **ungetc** returns *c*, and leaves the *stream* file unchanged.

If the *c* parameter is **EOF**, then the **ungetc** subroutine does not place anything in the buffer and a value of **EOF** is returned.

You can always push one character back onto a stream, provided that something has been read from the stream or **setbuf** has been called. The **fseek** subroutine erases all memory of inserted characters.

The **ungetc** subroutine returns **EOF** if it cannot insert the character.

Related Information

In this book: “fseek, rewind, ftell” on page 3-196, “getc, fgetc, getchar, getw” on page 3-204, “setbuf” on page 3-330, and “standard i/o library” on page 3-342.

varargs

Purpose

Handles a variable-length parameter list.

Syntax

#include <varargs.h>

va_alist

va_dcl

void va_start (*argp***)**
va_list *argp***;**

type **va_arg (***argp***,** *type***)**
va_list *argp***;**

void va_end (*argp***)**
va_list *argp***;**

Description

This set of macros allows you to write portable subroutines that accept a variable number of parameters. Subroutines that have variable-length parameter lists (such as **printf**), but that do not use **varargs**, are inherently nonportable because different systems use different parameter-passing conventions.

va_alist Is used as the parameter list in the function header.

va_dcl Is the declaration for **va_alist**. No semicolon should follow **va_dcl**.

va_list Defines the type of the variable used to traverse the list.

va_start Initializes *argp* to point to the beginning of the list.

argp Is a variable that the **varargs** macros use to keep track of the current location in the parameter list. Do not modify this variable.

va_arg Returns the next parameter in the list pointed to by *argp*. *type* is the data type that the parameter is expected to be. Different types can be mixed, but your subroutine must know what type of parameter is expected because it cannot be determined at runtime. The **printf** subroutine solves this problem by using its *format* parameter to determine the parameter types expected.

var_end Cleans up at the end.

Your subroutine can traverse, or scan, the parameter list more than once. Start each traversal with a call to **va_start** and end it with **var_end**.

Note: The calling routine is responsible for specifying the number of parameters because it is not always possible to determine this from the stack frame. For example, **execl** is passed a **NULL** pointer to signal the end of the list. **printf** determines the number of parameters from its *format* parameter.

Specifying **char**, **short**, or **float** as the second parameter to **va_arg.** is not portable because parameters seen by the called subroutine are not **char**, **short**, or **float**. The C complier converts **char** and **short** parameters to **int**, and it converts **float** parameters to **double** before passing them to a subroutine.

Example

The following example is a possible implementation of **execl** system call:

```
#include <varargs.h>

#define MAXARGS 100
/*
**  execl is called by
**  execl(file, arg1, arg2, . . . , (char *) 0);
*/
execl(va_alist)
   va_dcl
{
   va_list ap;
   char *file;
   char *args[MAXARGS];
   int argno = 0;

   va_start(ap);
   file = va_arg(ap, char *);
   while ((args[argno++] = va_arg(ap, char *)) != (char *) 0)
      ;   /* Empty loop body */
   va_end(ap);
   return (execv(file, args));
}
```

Related Information

In this book: “exec: execl, execv, execle, execve, execlp, execvp” on page 2-34, “printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf” on page 3-300, and “vprintf, vfprintf, vsprintf” on page 3-374.

vprintf, vfprintf, vsprintf

Purpose

Formats a **varargs** parameter list for output.

Library

Standard I/O Package (**libc.a**)

Syntax

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

int vprintf (format, argp)
char *format;
va_list argp;

int vfprintf (stream, format, argp)
FILE *stream;
char *format;
va_list argp;

int vsprintf (s, format, argp)
char *s, *format;
va_list argp;
```

Description

The **vprintf**, **vfprintf**, and **vsprintf** subroutines format and write **varargs** parameter lists. They are the same as the **printf**, **fprintf**, and **sprintf** subroutines, respectively, except that they are not called with a variable number of parameters. Instead, they are called with a parameter list pointer as defined by "varargs" on page 3-371.

Example

The following example demonstrates how the **vfprintf** subroutine could be used to write an error routine.

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

/* error should be called with the syntax:       */
/* error(routine_name, format [, value, . . . ]); */
```

```
/*VARARGS0*/

void error(va_alist)
va_dcl
/*
**  Note that the function name and format arguments
**  cannot be separately declared because of the
**  definition of varargs.
*/
{
   va_list args;
   char *fmt;

   va_start(args);
   /*
   ** Display the name of the function that called error
   */
   (void) fprintf(stderr, "ERROR in %s: ", va_arg(args, char *));
   /*
   ** Display the remainder of the message
   */
   fmt = va_arg(args, char *);
   (void) vfprintf(fmt, args);
   va_end(args);
   (void) abort();
}
```

Related Information

In this book: "printf, fprintf, sprintf, NLprintf, NLfprintf, NLsprintf" on page 3-300.

vrcppr

Purpose

Installs or removes a protocol procedure.

Library

Run-time Services Library (**librts.a**)

Syntax

#include <vrcppr.h>

int vrcppr (*request*, *path***)**
char *request*, **path*;

Description

The **vrcppr** subroutine issues a **DEFINE_CODE** supervisor call to add or delete a protocol procedure from Virtual Resource Manager (VRM). A ***protocol procedure*** is a special-purpose program that runs in VRM to support an application or service running on AIX.

The calling process's effective user ID must be superuser to use the **vrcppr** subroutine.

The value of the *request* parameter determines whether to add or delete the protocol procedure:

`'a'` Adds the protocol procedure that is contained in the file named by the *path* parameter. The file must be an executable file in **a.out** format.

`'d'` Deletes the protocol procedure named by the *path* parameter.

In both cases the *path* parameter must specify the full path name.

The **/etc/system** and **/etc/master** files must be set up correctly for the **vrcppr** subroutine to work properly. The **/etc/system** file must contain a stanza with the following information:

sstname:	The name of this **/etc/system** stanza.
`protocol  = true`	This is a protocol procedure.
`nospecial = true`	No **/dev** special file is to be created.
`noshow    = true`	The **devices** command should not display this device
`driver    =` *mstname*	The name of the corresponding stanza in **/etc/master**.

The **/etc/master** file must contain a corresponding stanza with the following information:

mstname:	The name of this **/etc/master** stanza.
`protocol = true`	This is a protocol procedure.
`code     =` *path*	The full path name of the protocol procedure.
`iocn     =` *iocn*	The IOCN for this code.

The value of the **code** keyword must be the same as the *path* parameter passed to the **vrcppr** subroutine.

Return Value

Upon successful completion, a nonnegative integer value representing the IOCN of the installed protocol procedure is returned. If the **vrcppr** subroutine fails, then it returns one of the following negative values:

VRM_inaa The calling process's effective user ID is not superuser.

VRM_pnnf The file named by the *path* parameter does not appear as the value of a **code** keyword in the **/etc/master** file.

VRM_snnf The file named by the *path* parameter does not appear as the name of stanza in the **/etc/system** file.

VRM_mopn The **/etc/master** file cannot be opened.

VRM_sopn The **/etc/system** file cannot be opened.

VRM_defpp The **vrmconfig** program failed.

VRM_bopt The *request* parameter is not `'a'` or `'d'`.

VRM_fork The **fork** system call failed.

VRM_pnam The *path* parameter does not specify a valid full path name.

VRM_iocn The stanza of the **/etc/master** file that contains **code** = *path* does not contain definition for the **iocn** keyword.

FORKERR The **fork** system call failed.

EXECERR The **exec** system call failed.

Files

/etc/master
/etc/system
/etc/vrmconfig

Related Information

In this book: "attributes" on page 4-20, "master" on page 4-98, "system" on page 4-139, and "config" on page 6-7.

Index

Special Characters

.init.state file format 4-3
_C_ prefix 3-129, 3-336, 5-60
_C_func 3-129, 3-336, 5-60
_exit system call 2-40
_NCtolower macro 3-39
_NCtoupper macro 3-39
_NCxcol macro 3-267
_NLxcol macro 3-267
_tolower subroutine 3-39
_toupper subroutine 3-39

A

a.out file 4-5
a.out relocation 4-9
a.out segments
  data 4-5
  stack 4-5
  text 4-5
a.out structure 4-5
abort subroutine 3-5
abs subroutine 3-6
absolute value function 3-167
absolute value, integer 3-6
access
  exclusive 2-64
access list
  group 2-126, 3-230
access system call 2-9
access time
  file 2-180
access utmp file entry 3-224
accessibility, determine file 2-9
accounting
  process 2-11
accounting file structure 4-15
accounting, process file 4-15
acct file 4-15
acct system call 2-11
acos subroutine 3-335
action
  upon receipt of signal 2-145
acute accent character 5-10
add a device 3-15
add a minidisk 3-19
add protocol procedure 3-376
addch subroutine 3-52, 3-134
addressing
  kernel mode 1-14
  user mode 1-12
addstr subroutine 3-52, 3-135
Advanced Floating-Point Accelerator 3-170, 3-183
afork flag 4-16
AIX file system 1-22
AIX kernel 1-6
AIX kernel, rebuild 3-21
AIX system name
  extended 2-172
  get 2-172
AIX trace collector 3-362
alarm clock
  set 2-13
alarm system call 2-13
allocating free blocks 1-29
allocation
  change data segment space 2-14
  free blocks 1-29
  i-number 1-28
allocator, main memory 3-236
ANSI floating point 3-170
APC/881 3-170, 3-183, 3-190
append
  data to a file 2-184
apply configuration information 3-21

ar file 4-18
arc subroutine 3-296
arccosine function 3-335
archive file format 4-18
archive file member structure 4-18
archive format, cpio 4-41
arcsine function 3-335
arctangent function 3-335
argc parameter 2-35
argument list, print 3-374
argv parameter 2-35
ASCII character set 5-3
ASCII controls 5-11
ASCII facility 5-3
ASCII to floating-point conversion 3-8
ASCII to integer conversion 3-4
asctime subroutine 3-46
asin subroutine 3-335
assembler output file 4-5
assert subroutine 3-7
assertion verification 3-7
assign a DOS Services drive 3-70
assign buffering to a stream 3-330
atan subroutine 3-335
atan2 subroutine 3-335
atof subroutine 3-8
atoi subroutine 3-347
atol subroutine 3-347
atomic operation 2-150
attach
  mapped file 2-131
  shared memory segment 2-131
attribute file 3-23
attribute file, close 3-25
attribute file, read stanza 3-31
attribute files 3-27, 3-29
attributes
  file system 4-64
attributes file 4-20
attroff subroutine 3-52
attron subroutine 3-52
attrset subroutine 3-52
automatic new line mode (AUTONL) 6-69
AUTONL mode 6-69
a64l subroutine 3-4

B

backend
  burst pages B-3
  exit codes B-7
  extra print copies B-6
  job charge B-6
  job status information B-6
  return error messages B-7
  routines in libqb B-8
  SIGTERM terminate B-8
  waiting state B-8
backends B-1
backup file 4-23
baudrate subroutine 3-52
beep subroutine 3-52, 3-136
bessel subroutines 3-9
bffree kernel subroutine C-29
bfget kernel subroutine C-28
binary input/output 3-192
binary search 3-11
binary search trees 3-364
binary synchronous communications 6-11
BIOCA C-28
BISYNC 6-11
block
  signal delivery 2-143
block I/O communication area (BIOCA) C-28
block 0 layout 1-25
blocked signals
  release 2-150
blocks
  allocation of free 1-29
  data 1-28
  delayed 2-163
  free 1-28
  superblock 1-25
bootstrap 1-9, 4-3
box subroutine 3-53, 3-136
brelse kernel subroutine C-28
breve accent character 5-10
brk system call 2-14
BRKINT 6-117

bsc device driver 6-11
BSDLY 6-118
bsearch subroutine 3-11
BS0 6-118
BS1 6-118
buffer subsystem 1-36
buffered I/O 3-342
buffering assignment to a stream 3-330
buffers C-15
build kernel C-51
bus
  I/O 1-37
bus special file 6-5
byte swapping 3-349

C

_C_ prefix 3-129, 3-336, 5-60
_C_func 3-129, 3-336, 5-60
caddr_t data type 5-75
call switch table 1-36
calling sequence v
calloc subroutine 3-236
calls
  to devices 1-40
calls, AIX supervisor
  See system calls
calls, function
  See kernel subroutines
  See subroutines
calls, kernel
  See kernel subroutines
  See system calls
calls, routine
  See kernel subroutines
  See subroutines
calls, subroutine
  See kernel subroutines
  See subroutines
calls, supervisor, AIX
  See system calls
calls, system
  See system calls
cancel sound 6-67
caron accent character 5-10
case
  conversion 3-39, 3-276, 3-278
  translation 3-39, 3-276
CBAUD 6-119
cbox subroutine 3-136
cbreak subroutine 3-53
cc.cfg file 4-29
cedilla accent character 5-10
ceil subroutine 3-167
ceiling function 3-167
cfgabdds subroutine 3-13
cfgadev subroutine 3-15
cfgamni subroutine 3-19
cfgaply subroutine 3-21
cfgcadsz subroutine 3-23
cfgcclsf subroutine 3-25
cfgcdlsz subroutine 3-27
cfgcopsf subroutine 3-29
cfgcrdsz subroutine 3-31
cfgddev subroutine 3-33
cfgdmni subroutine 3-36
change
  access permissions 2-18
  current directory 2-16
  data segment space allocation 2-14
  effective root directory 2-23
  file mode 2-18
  group of a file 2-21
  owner of a file 2-21
change current DOS Services directory 3-72
change current DOS Servicesdrive 3-72
change DOS file mode 3-74
change fonts 6-71
change modification date of DOS file 3-108
change priority
  of a process 2-88
channel
  create 2-95
character
  conversion 3-278
  lists C-25
  single shift 5-9
  two-byte 5-9
character classification 3-49
  international character support 3-270

character code processing 6-69
character codes 5-24
character collation
  code point 3-280
  international character support 3-267
character I/O 3-369
character set
  ASCII 5-3
character set definition 6-69
character translation 3-39, 3-276
character, get from stream 3-204
characteristics
  virtual machine 1-3
characteristics, device 4-57
characters
  international character support 3-276
characters, nonspacing 5-10
characters, two-byte 5-9
chdir system call 2-16
check whether trace channel is enabled 3-357
chgat subroutine 3-136
child process 1-17, 2-46
  control 2-102
  wait for termination of 2-182
child process times
  get 2-165
chmod system call 2-18
chown system call 2-21
chownx system call 2-21
chroot system call 2-23
circle subroutine 3-296
circumflex accent character 5-10
classify characters 3-49
clear subroutine 3-53, 3-137
clearerr macro 3-165
clearok subroutine 3-53, 3-137
clists C-25
CLOCAL 6-120
clock
  set alarm 2-13
clock rate 2-165
clock resolution 3-38
clock subroutine 3-38
close
  a file 2-25
close a stream 3-163
close all files 3-112
close an attribute file 3-25
close routine (ddclose) C-7
close system call 2-25
closepl subroutine 3-296
closing a DOS file 3-75
clrtobot subroutine 3-53, 3-137
clrtoeol subroutine 3-53, 3-137
cnt_t data type 5-75
code page 5-5, 5-6, 5-7, 5-8, 5-9
  P0 5-6, 5-25
  P1 5-7, 5-33
  P2 5-8, 5-40
  switching 5-9
code point 5-5
  character collation 3-280
collector, AIX errors 3-126
color palette, setting 6-70
colorend subroutine 3-137
colorout subroutine 3-138
COLUMNS variable 3-353
communication, interprocess 2-5, 3-198
communications 6-11
compile regular expression 3-318
complementary error function 3-125
config device driver 6-7
config device driver structure 6-9
config disk structure 6-8
configuration information, apply 3-21
connect.con file 4-33
construct a unique file name 3-247
construct the name for a temporary file 3-355
cont subroutine 3-296
contents
  directory 1-30
control
  execution of child process 2-102
  file 2-44
  I/O devices 2-56
control characters 5-11
control operations
  shared memory 2-135
control registers
  virtual machine 1-4
control sequence, virtual terminal data 6-61
control sequences 5-13

controlling terminal interface 6-131
controls 5-10
conversion subroutines 3-276
convert
  ASCII string to floating-point number 3-8
convert base-64 ASCII to long integer 3-4
convert between 3-byte integers and long integers 3-232
convert date and time to string 3-46
convert floating-point number to string 3-121
convert formatted input 3-325
convert long integer to base-64 ASCII string 3-4
convert string to integer 3-347
copyin kernel subroutine C-14
copyout kernel subroutine C-14
core file 4-39
cos subroutine 3-335
cosh subroutine 3-337
cosine function 3-335
costomize files C-45
cpass kernel subroutine C-12
cpio file 4-41
cpio structure 4-41
CPU time used report 3-38
CRDLY 6-118
CREAD 6-120
creat system call 2-27
create
  interprocess channel 2-95
  new file 2-27
  new process 2-46
create a directory 3-90
create a DOS temporary file 3-92
create a temporary file 3-354
create_ipc_prof subroutine 3-40.2
creating a DOS file 3-76
creating backends B-1
cresetty subroutine 3-138
crmode subroutine 3-138
crypt subroutine 3-42
CR0 6-118
CR1 6-118
CR2 6-118
CR3 6-118
csavetty subroutine 3-138
CSIZE 6-120
CSTOPB 6-120
ctermid subroutine 3-44
ctime subroutine 3-46
ctype macros 3-49
current directory, full path name 3-96
current directory, get path name of 3-206
current directory, path name 3-96
current DOS Services directory, change 3-72
current DOS Services drive, change 3-72
current signal mask
  setting 2-152
curses subroutine library 3-51
cursor representation 6-72
cuserid subroutine 3-62
customize
  files C-45
  helper program C-50
customize file format C-46
customize files
  /etc/ddi C-48
  /etc/master C-47
  /etc/system C-46
  relationships C-48
customize helper 3-13
customize helper program C-50

D

daddr_t data type 5-75
daemon, trace 3-359
data
  append to a file 2-184
  lock 2-97
  unlock 2-97
data access
  machine-independent 3-334
data base subroutines
data base, terminal capability 4-148
data blocks 1-28
data segment 1-12
  change space allocation 2-14
data stream
  3270 6-11

data structures
  file system 1-33
  I/O 1-38
data types, defined 5-75
data types, major
  monitor mode 6-73
date format 3-288
date to string conversion 3-46
date, modification, change 3-108
daylight external variable 3-46
dbm subroutines 3-63
dbminit subroutine 3-63
ddclose routine C-7
ddi 4-56, 4-110
ddi file 4-43
ddinit routine C-6
ddintr routine C-9, C-18
ddioctl routine C-8
ddopen routine C-6
ddread routine C-10
DDS 3-13
ddselect routine C-11
ddstrategy routine C-17
ddwrite routine C-10
declarations, parameter v
Define_Code SVC 6-7
define_device structure 3-13
define_device SVC 3-13
del_ipc_prof subroutine 3-64.1
delay kernel subroutine C-22
delay_output subroutine 3-53, 3-57
delch subroutine 3-53, 3-139
delete a device 3-33
delete a DOS file 3-110
delete a minidisk 3-36
delete protocol procedure 3-376
delete stanza 3-27
deleteln subroutine 3-53, 3-139
delta table format 4-136
delwin subroutine 3-53, 3-139
description file, port 4-117
description, file system 4-64
descriptions file format 4-56
descriptor
  file 2-111
detach
  shared memory segment 2-138
dev_t data type 5-75
device characteristics 4-57
device-dependent information 4-56, 4-110
device driver 1-39
  entry points C-3
  interface routines C-3
  kernel 1-36
  VRM 1-36
device driver error log C-31
device driver trace C-32
device drivers
  See also special files
  definition 1-40
  trace 6-128
device drivers, installing C-45
device drivers, writing C-1
device I/O 1-40
device interrupt handler C-9, C-18
device management 1-39
device number
  major 1-39
  minor 1-39
device status, DOS Services 3-114
device switch table 1-36, C-3
device, add 3-15
device, delete 3-33
devices
  See special files
devinfo structure 4-57, 6-100
devsw table C-3
DFT 6-11
dft device driver 6-11
diacritic characters 5-10
dir file 4-60
direct path 1-30
directory
  change current 2-16
  change the root 2-23
  create 2-69
directory change, DOS Services 3-72
directory creation 3-90
directory entry 4-60
  create a new 2-62
  remove 2-174
directory entry ".." 4-60

directory entry "." 4-60
directory file 1-23
directory file structure 4-60
directory format 4-60
directory removal, DOS Services 3-102
directory, full path name of current 3-96
directory, path name of current 3-206
directoy contents 1-30
disclaim system call 2-30
disk buffer headers C-27
disk buffers C-15
diskette file 6-17
diskette structure 6-17
display symbols 5-24
display, changing physical 6-67
dispsym definition 5-24
distance function, euclidean 3-229
Distributed Function Terminal 6-11
DOS Services assign 3-70
DOS Services directory, change 3-72
DOS Services drive, change 3-72
DOS Services environment, initialize 3-85
DOS Services file handle duplication 3-78
DOS Services program execution 3-79
DOS Services subroutine library 3-65
DOS Servicesdirectory removal 3-102
DOS file access 3-65
DOS file creation 3-76
DOS file lock 3-88
DOS file mode, change 3-74
DOS file modification date, change 3-108
DOS file read 3-98
DOS file read/write pointer, move 3-104
DOS file rename 3-100
DOS file status, get 3-106
DOS file system D-1
DOS file write 3-116
DOS file, close 3-75
DOS file, delete 3-110
DOS file, open 3-94
DOS file, unlink 3-110
DOS files synchronization 3-83
DOS function call table D-2
DOS function calls D-2
DOS programs, porting D-1
DOS temporary file creation 3-92
dosassign subroutine 3-70
doschdir subroutine 3-72
doschmod subroutine 3-74
dosclose subroutine 3-75
doscreate subroutine 3-76
dosdup subroutine 3-78
dosexecve subroutine 3-79
dosfirst subroutine 3-81
dosfstat subroutine 3-106
dosfsync subroutine 3-83
dosinit subroutine 3-85
doslock subroutine 3-88
dosmkdir subroutine 3-90
dosmktemp subroutine 3-92
dosnext subroutine 3-81
dosopen subroutine 3-94
dospwd subroutine 3-96
dosread subroutine 3-98
dosrename subroutine 3-100
dosreopen subroutine 3-112
dosrmdir subroutine 3-102
dosseek subroutine 3-104
dosstat subroutine 3-106
dostouch subroutine 3-108
dosunlink subroutine 3-110
dosunopen subroutine 3-112
dosustat subroutine 3-114
doswrite subroutine 3-116
double acute accent character 5-10
doupdate subroutine 3-53
drand48 subroutine 3-118
drawbox subroutine 3-139
drive change, DOS Services 3-72
drive, DOS Services, assign 3-70
driver format, message 6-105
driver, event-tracing 6-128
drivers
  hft 6-23
drivers, device
  See special files
drivers, writing device C-1
drsname subroutine 3-120.1
drsnidd subroutine 3-120.1
dsstate system call. 2-30.2
dup system call 2-32
duplicate an open file descriptor 2-32

duplicating a DOS Services file handle 3-78

E

EBCDIC character set 5-45
ecactp subroutine 3-140
ecadpn subroutine 3-140
ecaspn subroutine 3-140
ecblks subroutine 3-140
ecbpls subroutine 3-141
ecbpns subroutine 3-141
ecdfpl subroutine 3-142
ecdppn subroutine 3-143
ecdspl subroutine 3-143
ecdvpl subroutine 3-144
ecflin subroutine 3-145
ECHO 6-120
echo subroutine 3-53, 3-147
ECHOE 6-121
ECHOK 6-121
ECHONL 6-121
ecpnin subroutine 3-147
ecrfpl subroutine 3-148
ecrfpn subroutine 3-148
ecrlpl subroutine 3-148
ecrmpl subroutine 3-149
ecscpn subroutine 3-149
ecshpl subroutine 3-149
ectitl subroutine 3-150
ecvt subroutine 3-121
edata 3-123
emulation, hft 6-54
encrypt subroutine 3-42
encrypted password 4-113
encryption, password 3-42
end 3-123
endgrent subroutine 3-210
endpwent subroutine 3-219
endutent subroutine 3-224
endwin subroutine 3-53, 3-150
enhanced signal facilities 2-156
entries in name list, obtaining 3-283
entry points, device driver C-3
environ global variable 2-35
environment 2-35
environment alteration 3-310.1
environment facility 5-47
environment setting 4-127
environment subroutines 3-208, 3-280
  getenv 3-208
  NLgetenv 3-208
environment variable, value of 3-208
environment, initialize DOS Services 3-85
envp parameter 2-35
eof character 6-115
eol character 6-115
eqn special character definitions 5-54
eqnchar facility 5-54
erand48 subroutine 3-118
erase
  portion of a file 2-42
erase character 6-115
erase subroutine 3-53, 3-150, 3-296
erasechar subroutine 3-53
erf subroutine 3-125
erfc subroutine 3-125
errfile file 4-62
errno 3-294
errno values A-1
errno.h A-1
error codes A-1
error collector, AIX 3-126
error file 6-15
error function 3-125
error-handling function 3-238
error log, device driver C-31
error log, kernel C-31
error logging 6-15
error messages 3-294
error numbers A-1
error values A-1
errprintf kernel subroutine C-30
errsave kernel subroutine C-31
errunix subroutine 3-126
escape sequences 5-13
etext 3-123
euclidean distance function 3-229
event log file 4-62
event logging 6-15
event-tracing driver 6-128

exception handling, floating-point 3-188
exclusive access
  to a file region 2-64
exec system call 2-34
execl system call 2-34
execle system call 2-34
execlp system call 2-34
execute
  file 2-34
execute a program with a DOS path name 3-79
execution monitor 3-248
execution profile 3-248
execution suspension 3-338
execution time
  profile 2-99
execv system call 2-34
execve system call 2-34
execvp system call 2-34
exit system call 2-40
_exit system call 2-40
exp subroutine 3-128
exponential function 3-128
exponentiation 3-128
expression, regular 3-318, 3-321
extended AIX system name 2-172
extended curses subroutine library 3-131
extended message receive 2-85
extended path name C-20
extended read 2-106
extended subroutine 3-150
externals
  edata 3-123
  end 3-123
  etext 3-123

F

F_DUPFD 2-44
F_GETFD 2-44
F_GETFL 2-44.1
F_GETLK 2-44.1
F_SETFD 2-44.1
F_SETFL 2-44.1
F_SETLK 2-44.2
F_SETLKW 2-44.2
fabs subroutine 3-167
facilities
  mm 5-62
  regexp 3-321
facilities, miscellaneous
  See miscellaneous facilities
fault generation, IOT 3-5
fclear system call 2-42
fclose subroutine 3-163
fcntl system call 2-44
fcntl.h header file 5-56
fcvt subroutine 3-121
fd devinfo structure 6-18
fd file 6-17
fdopen subroutine 3-168
feof macro 3-165
ferror macro 3-165
fetch subroutine 3-63
FFDLY 6-118
fflush subroutine 3-163
ffullstat system call 2-50.2
FF0 6-118
FF1 6-118
fgetc subroutine 3-204
fgets subroutine 3-221
fifo
  create 2-69
file 2-90, 2-106
  accessibility, determine 2-9
  close a 2-25
  control 2-44
  create 2-69
  creation 2-27
  directory entry
    create a new 2-62
  erase portion of 2-42
  execute 2-34
  lock a region 2-64
  mode change 2-18
  open to read or write 2-90
  read from 2-106
  read from, extended 2-106
  rewrite 2-27
  shorten 2-50
  unlock a region 2-64

write 2-184
write changes 2-48
file access
set time 2-180
file control 2-3
file creation mask
get 2-169
set 2-169
file creation, DOS, temporary 3-92
file creation, temporary 3-354
file descriptor 2-111
close 2-25
duplication 2-32
file entry, group, obtaining 3-210
file entry, utmp access 3-224
file formats
archive 4-18
process accounting 4-15
file i/o subsystem 1-36
file maintenance 2-3
file mapping 2-7
file member, archive structure 4-18
file mode change, DOS 3-74
file modification
set time 2-180
file name generation, terminal 3-44
file name, construct 3-247
file name, make 3-247
file naming, temporary files 3-355
file pointer
read/write 2-67
file pointer repositioning 3-196
file status
obtain 2-159
file status, DOS, get 3-106
file synchronization, DOS 3-83
file system
backup format 4-23
data structures 1-33
DOS D-1
layout 1-25
mount 2-71
statistics 2-178
unmount 2-170
file system attributes 4-64
file system description 4-64
file system management 1-22
file system table 4-108
file tree, read 3-200
file types 1-23
directory 1-23
ordinary 1-24
special 1-24
file, assembler output 4-5
file, link editor output 4-5
file, storage image 4-39
fileno macro 3-165
files
directory 2-69
header vii
mapped 2-7
ordinary 2-69
special 1-40, 2-69, 2-71
files, device
See special files
files, special
See special files
filesystems file 4-64
find DOS files that match a pattern 3-81
find slot in utmp file for current user 3-368
find value of user information name 3-223
find_ipc_prof subroutine 3-166.1
firstkey subroutine 3-63
fixterm subroutine 3-53
flag letter, get from argument vector 3-214
flash subroutine 3-53, 3-150
floating-point
conversion from ASCII 3-8
Floating-Point Accelerator 3-170, 3-183
floating-point exception handling 3-188
floating-point numbers manipulation 3-194
floating-point subroutines, ANSI/IEEE 3-170
floating-point to string conversion 3-121
floor function 3-167
floor subroutine 3-167
flush a stream 3-163
flushinp subroutine 3-53
fmod subroutine 3-167
font file format 4-68
font symbols 5-24
fonts, changing 6-71
fopen subroutine 3-168

fork 2-46
form v
format v, 3-288
  date 3-288
  time 3-288
format of cpio archive 4-41
format of SCCS file 4-135
format specification, text files 4-82
format, archive 4-18
format, gps 4-84
format, message driver 6-105
format, system volume 4-74
formats
  directory 4-60
  event log file 4-62
  inode 4-92
  master 4-98
  SCCS delta table 4-136
  SCCS file 4-135
formats, file
  See file formats
formatted input conversion 3-325
formatted output, print 3-300
formatted varargs argument list, print 3-374
formatting a permuted index, macro package 5-63
FP_DOUBLE 3-170
FP_FLOAT 3-170
fpfp subroutines 3-170
fprintf subroutine 3-300
fputc subroutine 3-309
fputs subroutine 3-313
fread subroutine 3-192
free-block list 1-28
free blocks
  allocation 1-29
free kernel subroutine C-25
free subroutine 3-236
freopen subroutine 3-168
frexp subroutine 3-194
fs file 4-74
fscanf subroutine 3-325
fseek subroutine 3-196
fspec file 4-82
fstat system call 2-159
fsync system call 2-48
ftell subroutine 3-196
ftok subroutine 3-198
ftruncate system call 2-50
ftw subroutine 3-200
fubyte kernel subroutine C-14
fullbox subroutine 3-151
fullstat structure 5-56.2
fullstat system call 2-50.2
fullstat.h header file 5-56.2
function calls
  DOS D-2
function libraries
  See libraries
function, complementary error 3-125
function, error 3-125
function, error-handling 3-238
function, euclidean distance 3-229
functions
  See also kernel subroutines
  See also subroutines
  absolute value 3-167
  ceiling 3-167
  floor 3-167
  remainder 3-167
functions hyperbolic 3-337
functions, trigonometric 3-335
fuword kernel subroutine C-14
fwrite subroutine 3-192

G

gamma function 3-202
gamma subroutine 3-202
gcvt subroutine 3-121
generate file name for terminal 3-44
generate pseudo-random numbers 3-317
generating an IOT fault 3-5
geometric text font 4-72.4
get
  file status 2-159
  group IDs 2-55
  message queue identifier 2-76
  process IDs 2-54
  time 2-164

  user IDs 2-55
get a string from a stream 3-221
get character or word from stream 3-204
get DOSfile status 3-106
get file system statistics 2-178
get group file entry 3-210
get login name 3-212
get names from name list 3-283
get option letter from argument vector 3-214
get password file entry 3-219
get path name of current directory 3-206
get status of DOS Services device 3-114
get the name of a terminal 3-367
get user name 3-62
getc kernel subroutine C-26
getc macro 3-204
getcb kernel subroutine C-27
getcf kernel subroutine C-26
getch subroutine 3-53, 3-151
getchar macro 3-204
getcwd subroutine 3-206
geteblk kernel subroutine C-28
getegid system call 2-55
getenv subroutine 3-208
geteuid system call 2-55
getgid system call 2-55
getgrent subroutine 3-210
getgrgid subroutine 3-210
getgrnam subroutine 3-210
getgroups system call 2-52
getlogin subroutine 3-212
getopt subroutine 3-214
getpass subroutine 3-217
getpgrp system call 2-54
getpid system call 2-54
getppid system call 2-54
getpw subroutine 3-218
getpwent subroutine 3-219
getpwnam subroutine 3-219
getpwuid subroutine 3-219
gets subroutine 3-221
getstr subroutine 3-53, 3-152
gettmode subroutine 3-53, 3-152
getuid system call 2-55
getuinfo subroutine 3-223
getutent subroutine 3-224
getutid subroutine 3-224
getutline subroutine 3-224
getw subroutine 3-204
getyx subroutine 3-53, 3-152
gmtime subroutine 3-46
goto, nonlocal 3-332
gps format 4-84
graphic output file format 4-115
graphic symbols 5-24
graphics interface 4-115
graphics interface subroutines
grave accent character 5-10
Greek characters 5-58
greek facility 5-58
group access list 3-230
  get 2-52
  set 2-126
group file 4-87
group file entry, obtaining 3-210
group ID
  set 2-129
  set for a process 2-128
group ID of a file
  change 2-21
group ID translation 2-21
group IDs
  get 2-55
gsbply subroutine 7-20
gscarc subroutine 7-22
gscatt subroutine 7-24
gsccnv subroutine 7-26
gscir subroutine 7-29
gsclrs subroutine 7-31
gscmap subroutine 7-32
gscrca subroutine 7-34
gsdpik subroutine 7-36
gseara subroutine 7-38
gsearc subroutine 7-40
gsecnv subroutine 7-42
gsecur subroutine 7-45
gsell subroutine 7-46
gsepik subroutine 7-48
gseply subroutine 7-50
gsevds subroutine 7-52
gseven subroutine 7-54
gsevwt subroutine 7-56

gsfatt subroutine 7-63
gsfci subroutine 7-65
gsfell subroutine 7-67
gsfply subroutine 7-69
gsfrec subroutine 7-71
gsgtat subroutine 7-73
gsgtxt subroutine 7-78
gsignal subroutine 3-340
gsinit subroutine 7-80
gslatt subroutine 7-84
gslcat subroutine 7-86
gsline subroutine 7-88
gslock subroutine 7-90
gslop subroutine 7-92
gslpat subroutine 7-95
gsmask subroutine 7-97
gsmatt subroutine 7-99
gsmcur subroutine 7-102
gsmult subroutine 7-104
gspcls subroutine 7-106
gsplym subroutine 7-108
gspoly subroutine 7-110
gspp subroutine 7-112
gsqdsp subroutine 7-114
gsqfnt subroutine 7-117
gsqgtx subroutine 7-119
gsqloc subroutine 7-121
gsrrst subroutine 7-123
gsrsav subroutine 7-125
gstatt subroutine 7-128
gsterm subroutine 7-131
gstext subroutine 7-132
gsulns subroutine 7-134
gsunlk subroutine 7-136
gsvgrn subroutine 7-137
gsxblt subroutine 7-139
gsxcnv subroutine 7-146
gsxptr subroutine 7-148

H

handle, duplicating 3-78
handler, interrupt C-9, C-18
hardware access
  RT PC D-7
has_ic subroutine 3-53
has_il subroutine 3-53
hash tables 3-227
hcreate subroutine 3-227
HD devinfo structure 6-21
hdestroy subroutine 3-227
head, of screen manager ring 6-50
header files vii
help text, issue 3-252
help text, retrieve 3-263
helper, customize 3-13
hft device, query 3-352
hft driver 6-23
hft emulation 6-54
hft, remote 6-54
history file 4-89
hole
  make in a file 2-42
hsearch subroutine 3-227
HUPCL 6-120
hyperbolic cosine function 3-337
hyperbolic functions 3-337
hyperbolic sine function 3-337
hyperbolic tangent function 3-337
hypot subroutine 3-229

I

i-list layout 1-26
i-node layout 1-27
i-nodes
  update 2-163
i-number allocation 1-28
I/O 2-3
I/O activity
  wait for 2-111
I/O bus 1-37

I/O data structures 1-38
I/O devices
  See also special files
  control operations 2-56
I/O overview 1-34
I/O status
  check 2-111
I/O, buffered 3-342
ICANON 6-120
ICRNL 6-117
idlok subroutine 3-53
IEEE floating point 3-170
ieeetrap subroutine 3-189
IGNBRK 6-117
IGNCR 6-117
IGNPAR 6-117
ilog file 4-170
image, memory 6-103
image, virtual memory 6-103
immediate message, issue 3-255
inch subroutine 3-54, 3-152
init routine (ddinit) C-6
.init.state file format 4-3
initgroups subroutine 3-230
initial AIX state 4-3
initialize DOS Services environment 3-85
initialize group access list 3-230
initiate a pipe to or from a process 3-298
initscr subroutine 3-54, 3-153
INLCR 6-117
ino_t data type 5-75
inode format 4-92
inode structure 4-92
INPCK 6-117
input stream, put character back 3-369
input/output 2-3
input/output devices
  control operations 2-56
input/output, buffered 3-342
input/output, device 1-40
input, binary 3-192
inquiry, stream status 3-165
insch subroutine 3-54, 3-153
insert mode 6-69
insert, retrieve 3-263
insertln subroutine 3-54, 3-153
install protocol procedure 3-376
installing device drivers C-45
integer absolute value 3-6
integer from string conversion 3-347
integer to ASCII conversion 3-4
interface control, terminal 6-131
interface routines, device driver C-3
interface, graphics 4-115
international character support 3-288
  character classification 3-270
  character collation 3-267
  character conversion 3-39
  date format 3-288
  environment 3-280, 5-47
  formatted output 3-300
  NLchar data type 3-276
  parameter fetching 3-281
  string conversion 3-278
  string handling 3-288, 3-291
  string operations 3-272, 3-285
  time format 3-288
  time structure 3-291
interprocess channel
  create 2-95
interprocess communication 2-5, 3-198
interrupt handler C-9, C-18
interrupt handler. C-18
interrupt-level processing C-9, C-18
interrupt level, sublevel C-18
intr character 6-115
intr routine (ddintr) C-9, C-18
intrflush subroutine 3-54
ioctl routine (ddioctl) C-8
ioctl system call 2-56
iodone kernel subroutine C-17
iomove kernel subroutine C-13
IOT fault generation 3-5
IPC 2-5
ipc_perm structure 2-6
IPC_RMID 2-136
IPC_SET 2-136
IPC_STAT 2-135
IPL 4-3
  virtual machine 2-58, 2-109
iplvm system call 2-58
isalnum macro 3-49

isalpha macro 3-49
isascii macro 3-49
isatty subroutine 3-367
iscntrl macro 3-49
isdigit macro 3-49
isgraph macro 3-49
ISIG 6-120
islower macro 3-49
isprint macro 3-49
ispunct macro 3-49
isspace macro 3-49
issue a queued message 3-259
issue a shell command 3-350
issue an immediate message 3-255
issue help text 3-252
ISTRIP 6-117
isupper macro 3-49
isxdigit macro 3-49
IUCLC 6-117
IXANY 6-117
IXOFF 6-117
IXON 6-117

J

jrand48 subroutine 3-118
j0, j1, jn subroutines 3-9

K

kaf file format 4-94
kernel calls
  See kernel subroutines
  See system calls
kernel device driver 1-36
kernel error log C-31
kernel features 1-8
kernel functions
  file system management 1-7
  memory management 1-7
  process management 1-7
  program management 1-8
  resource management 1-8
  time management 1-7
kernel mode 1-10
kernel mode addressing 1-14
kernel rebuild C-51
kernel subroutines
  bffree C-29
  bfget C-28
  brelse C-28
  copyin C-14
  copyout C-14
  cpass C-12
  delay C-22
  errprintf C-30
  errsave C-31
  free C-25
  fubyte C-14
  fuword C-14
  getc C-26
  getcb C-27
  getcf C-26
  geteblk C-28
  iodone C-17
  iomove C-13
  kmsgctl C-24
  kmsgget C-24
  malloc C-25
  palloc C-25
  panic C-30
  passc C-13
  printf C-30
  psignal C-24
  putc C-26
  putcb C-27
  putcf C-27
  rmsgrcv C-24.1
  rmsgsnd C-24
  selwakeup C-12
  setmpx C-20
  sleep C-21
  splhi C-19
  splx C-19
  spl0—spl7 C-19
  subyte C-13
  suword C-13
  timeout C-21

trsave C-32
untimeout C-22
usrchar C-20
vec_clear C-18
vec_init C-18
wakeup C-21
kernel trace C-32
kernel trap routine 1-35
kernel, AIX, rebuild 3-21
key_t data type 5-75
keyboard 6-79
keypad subroutine 3-54, 3-153
keywords, ddi 4-56
kill character 6-115
kill system call 2-60
killchar subroutine 3-54
kmem file 6-103
kmsgctl kernel subroutine C-24
kmsgget kernel subroutine C-24

L

label subroutine 3-296
label_t data type 5-75
layout
block 0 1-25
file system 1-25
i-list 1-26
i-node 1-27
superblock 1-25
lcong48 subroutine 3-118
ldexp subroutine 3-194
leaveok subroutine 3-54, 3-154
letter, option, get from argument vector 3-214
level, interrupt C-18
level_t data type 5-75
lfind subroutine 3-234
libPW subroutine library 3-305
libraries
DOS Services 3-65
extended curses 3-131
programmers workbench 3-305
standard I/O 3-342
light-emitting diodes, setting 6-64
limits
user 2-167
line subroutine 3-296
linear congruential algorithm 3-118
linear search and update 3-234
linemod subroutine 3-296
LINES variable 3-352
link
create 2-62
link editor output file 4-5
link system call 2-62
list
free-block 1-28
lists
character C-25
loadtbl system call 2-62.2
localtime subroutine 3-46
locator thresholds 6-64
lock
data 2-97
process 2-97
region of a file 2-64
text 2-97
lock a region of a DOS file 3-88
lockf system call 2-64
log errors C-31
log subroutine 3-128
log trace entry C-32
logarithm 3-128
login name 3-62
login name of user, obtaining 3-233
login name, get 3-212
login, remote 6-107
logname subroutine 3-233
log10 subroutine 3-128
long integers from 3-byte integers 3-232
longjmp subroutine 3-332
longname subroutine 3-54, 3-154
lp special file 6-98
lprio structure 6-100
lprmode structure 6-100
LPRUDE structure 6-101
lrand48 subroutine 3-118
lsearch subroutine 3-234
lseek system call 2-67
ltol3 subroutine 3-232

l3tol subroutine 3-232
l64a subroutine 3-4

M

machine-independent data access 3-334
macro definitions vii
macro package for formatting a permuted index 5-63
macron accent character 5-10
macros
    _NCtolower 3-39
    _NCtoupper 3-39
    _tolower 3-39
    _toupper 3-39
    clearerr 3-165
    ctype 3-49
    feof 3-165
    ferror 3-165
    fileno 3-165
    getc 3-204
    getchar 3-204
    isalnum 3-49
    isalpha 3-49
    isascii 3-49
    iscntrl 3-49
    isdigit 3-49
    isgraph 3-49
    islower 3-49
    isprint 3-49
    ispunct 3-49
    isspace 3-49
    isupper 3-49
    isxdigit 3-49
    NCesc 3-39
    NCunesc 3-39
    putc 3-309
    putchar 3-309
    varargs 3-371
magic number 2-34
main memory allocator 3-236
main subroutine 2-35
maintenance 2-3
maintenance mode 4-3
make
    hole in a file 2-42
make a unique file name 3-247
malloc kernel subroutine C-25
malloc subroutine 3-236
management
    device 1-39
manipulate parts of floating-point numbers 3-194
mapped file
    attach 2-131
mapped files 2-7
mask
    file creation 2-169
master file 4-98
master format 4-98
match regular expression 3-318
math.h header file 5-60
matherr subroutine 3-238
mdverify subroutine 3-243
mem file 6-103
memccpy 3-245
memchr 3-245
memcmp 3-245
memcpy 3-245
memory allocator 3-236
memory control operations
    shared 2-135
memory image 6-103
memory image file 6-103
memory locations
    predefined 1-4
memory management 1-10
memory-mapped files 2-7
memory operations 3-245
memory segment
    attach to process 2-131
    detach 2-138
    get 2-140
memory subroutine 3-245
memory, disclaim 2-30
memset 3-245
message
    control operations 2-73
    from queue 2-79
message control 2-73

message driver format 6-105
message file 4-105
message queue 2-111
  get identifier 2-76
  send message 2-82
message queues C-24
message receive
  extended 2-85
message, issue a queued 3-259
message, issue an immediate 3-255
message, retrieve 3-263
messages, error 3-294
meta subroutine 3-54, 3-154
minidisk customizing 6-20
minidisk, add 3-19
minidisk, delete 3-36
minidisks 3-243
miscellaneous facilities
mkdir system call 2-68.1
mknod system call 2-69
mktemp subroutine 3-247
mm facility 5-62
mm macro package 5-62
mntctl system call 2-70.2
mnttab file 4-108
mnttab.h structure 4-108
mode bit
  set-group-ID 2-36
  set-user-ID 2-36
mode change, file 2-18
mode, DOS file, change 3-74
modes
  kernel 1-10
  user 1-10
modf subroutine 3-194
modification date, change, DOS file 3-108
modification time
  file 2-180
monitor mode major data type 6-73
monitor subroutine 3-248
mount
  file system 2-71
mount system call 2-71
mounted file system table 4-108
move
  read/write file pointer 2-67
move DOS file read/write pointer 3-104
move subroutine 3-54, 3-154, 3-296
mptx facility 5-63
mrand48 subroutine 3-118
msgbuf structure 2-79
msgctl system call 2-73, C-24
msgget system call 2-76, C-24
msghelp subroutine 3-252
msgimed subroutine 3-255
msgop system calls 2-79, 2-82, 2-85
msgqued subroutine 3-259
msgrcv system call 2-79
msgrtrv subroutine 3-263
msgsnd system call 2-82, C-24
msgxrcv system call 2-85, C-24
multi-byte characters 5-9
multi-byte controls 5-13
multi-user mode 4-3
multiplexed device C-20
Multiprotocol Adapter 6-11
mv facility 5-64
mvaddch subroutine 3-54, 3-134
mvaddstr subroutine 3-52, 3-135
mvchgat subroutine 3-136
mvcur subroutine 3-54, 3-155
mvdelch subroutine 3-54, 3-139
mvgetch subroutine 3-54, 3-151
mvgetstr subroutine 3-54, 3-152
mvinch subroutine 3-54, 3-152
mvinsch subroutine 3-54, 3-153
mvpaddch subroutine 3-134
mvpaddstr subroutine 3-135
mvpchgat subroutine 3-136
mvprintw subroutine 3-54
mvscanw subroutine 3-54
mvwaddch subroutine 3-52, 3-134
mvwaddstr subroutine 3-52, 3-135
mvwchgat subroutine 3-136
mvwdelch subroutine 3-54, 3-139
mvwgetch subroutine 3-54, 3-151
mvwgetstr subroutine 3-54, 3-152
mvwin subroutine 3-54, 3-155
mvwinch subroutine 3-54, 3-152
mvwinsch subroutine 3-54, 3-153
mvwprintw subroutine 3-55
mvwscanw subroutine 3-55

N

name for a temporary file, create 3-355
name list entries, obtaining 3-283
name of a terminal 3-367
name of the user 3-62
name, login 3-212
name, user login, obtaining 3-233
name, user, find value 3-223
NCchrlen macro 3-276
NCcollate subroutine 3-267
NCcoluniq subroutine 3-267
NCctype 3-270
NCdec macro 3-276
NCdechr macro 3-276
NCdecode subroutine 3-276
NCdecstr subroutine 3-276
NCenc macro 3-276
NCencode subroutine 3-276
NCencstr subroutine 3-276
NCeqvmap subroutine 3-267
NCesc macro 3-39
NCflatchar subroutine 3-39
NCisalnum subroutine 3-270
NCisalpha subroutine 3-270
NCisdigit subroutine 3-270
NCisgraph subroutine 3-270
NCislower subroutine 3-270
NCisNLchar subroutine 3-270
NCisprint subroutine 3-270
NCispunct subroutine 3-270
NCisshift subroutine 3-270
NCisspace subroutine 3-270
NCisupper subroutine 3-270
NCisxdigit subroutine 3-270
NCstrcat subroutine 3-272
NCstrchr subroutine 3-272
NCstrcmp subroutine 3-272
NCstrcpy subroutine 3-272
NCstrcspn subroutine 3-272
NCstring subroutine 3-272
NCstrlen subroutine 3-272
NCstrncat subroutine 3-272
NCstrncmp subroutine 3-272
NCstrncpy subroutine 3-272
NCstrpbrk subroutine 3-272
NCstrrchr subroutine 3-272
NCstrspn subroutine 3-272
NCstrtok subroutine 3-272
_NCtolower macro 3-39
NCtolower subroutine 3-39
NCtoNLchar subroutine 3-39
_NCtoupper macro 3-39
NCtoupper subroutine 3-39
NCunesc macro 3-39
_NCxcol macro 3-267
neqn special character definitions 5-54
new-line character 6-115
new process image 2-34
newpad subroutine 3-55
newterm subroutine 3-55
newview subroutine 3-155
newwin subroutine 3-55, 3-156
nextkey subroutine 3-63
nice system call 2-88
nl subroutine 3-55, 3-156
NLchar data type 3-276
NLchrlen macro 3-276
NLconvstr subroutines 3-278
NLDLY 6-118
NLecflin subroutine 3-145
NLescstr subroutine 3-278
NLflatstr subroutine 3-278
NLfprintf subroutine 3-300
NLfscanf subroutine 3-325
NLgetctab subroutine 3-280
NLgetenv subroutine 3-208
NLgetfile 3-281
NLisNLcp macro 3-276
nlist subroutine 3-283
NLO 6-118
NLprintf subroutine 3-300
NLscanf subroutine 3-325
NLsprintf subroutine 3-300
NLsscanf subroutine 3-325
NLstrcat subroutine 3-285
NLstrchr subroutine 3-285
NLstrcmp subroutine 3-285
NLstrcpy subroutine 3-285
NLstrcspn subroutine 3-285
NLstring 3-285

NLstrlen subroutine 3-285
NLstrncat subroutine 3-285
NLstrncmp subroutine 3-285
NLstrncpy subroutine 3-285
NLstrpbrk subroutine 3-285
NLstrrchr subroutine 3-285
NLstrspn subroutine 3-285
NLstrtime subroutine 3-288
NLstrtok subroutine 3-285
NLtmtime subroutine 3-291
NLunescstr subroutine 3-278
_NLxcol macro 3-267
NL1 6-118
nocbreak subroutine 3-53
nocrmode subroutine 3-138
nodelay subroutine 3-53, 3-156
noecho subroutine 3-53, 3-147
NOFLSH 6-121
nometa subroutine 3-154
non-standard tabbing 4-82
nonl subroutine 3-55, 3-156
nonlocal goto 3-332
nonspacing characters 5-10
noraw subroutine 3-55, 3-158
nrand48 subroutine 3-118
null special file 6-104
number
   magic 2-34
numbers, pseudo-random 3-118, 3-317
nvram file 6-103

O

OCRNL 6-118
OFDEL 6-118
OFILL 6-118
ogonek accent character 5-10
OLCUC 6-118
ONLCR 6-118
ONLRET 6-118
ONOCR 6-118
open a DOS file 3-94
open a stream 3-168
open attribute file 3-29
open file
   to read 2-90
   to write 2-90
open routine (ddopen) C-6
open system call 2-90
openpl subroutine 3-296
operating system profiler 6-106
operating system state 1-4
OPOST 6-118
oprmode structure 6-101
option letter, get from argument vector 3-214
options file format 4-110
ordinary file 1-24
os overview 1-3
osm driver 6-105
output file, assembler 4-5
output file, link editor 4-5
output, binary 3-192
output, print formatted 3-300
overcircle accent character
overdot accent character 5-10
overlay subrouDMS_tine 3-157
overlay subroutine 3-55
overview
   I/O 1-34
   signals 2-4
overview of system 1-3
overwrite subroutine 3-55, 3-157
owner ID translation 2-21
owner of a file 2-21
   change 2-21

P

paddch subroutine 3-134
paddr_t data type 5-75
paddstr subroutine 3-135
palette, setting color 6-70
palloc kernel subroutine C-25
panic kernel subroutine C-30
param.h header file 5-68
parameter passing 2-35
parameters v
PARENB 6-120

parent control
  of child process 2-102
parent directory 4-60
parent process 1-17, 2-46
parent process ID 2-54
PARMRK 6-117
PARODD 6-120
passc kernel subroutine C-13
passing
  parameter 2-35
passwd file 4-112
password description 4-113
password encryption 3-42
password file entry, get 3-218, 3-219
password file entry, write 3-312
password, read 3-217
path name
  direct 1-30
  relative 1-32
  resolution 1-30
path name extension C-20
path name of current directory 3-206
pattern, finding DOS files that match 3-81
pause system call 2-94
PC-DOS programs, porting D-1
pchgat subroutine 3-136
pclose subroutine 3-298
pcs font 4-72.4
perase subroutine 3-150
permanent storage
  write file to 2-48
permission
  file access 2-18
perror subroutine 3-294
physadr structure 5-75
pipe initiation 3-298
pipe system call 2-95
pixel map 7-142
plock system call 2-97
plot file format 4-115
plot subroutines 3-296
pnoutrefresh subroutine 3-55
point subroutine 3-296
pointer, DOS file read/write, move 3-104
popen subroutine 3-298
port description file 4-117

porting DOS 3.0 D-1
ports file 4-117
portstatus file 4-122
portstatus structure 4-122
pow subroutine 3-128
power (exponentiation) 3-128
predefined file 4-124
predefined memory locations 1-4
prefresh subroutine 3-55
prf file 6-106
primitive system data types 5-75
print
  formatted output 3-300
print floating-point number 3-121
print formatted varargs argument list 3-374
printf kernel subroutine C-30
printf subroutine 3-300
printw subroutine 3-55, 3-157
priority computation 1-20
priority of a process
  change 2-88
process
  child 1-17
  creation 2-46
  get IDs 2-54
  get owner 2-176
  lock 2-97
  parent 1-17
  preemption 1-19
  set owner 2-176
  states 1-19
  trace execution 2-102
  unlock 2-97
process accounting 2-11
process accounting file 4-15
process addressing 1-10
process alarm 2-13
process communication
  signals 1-20
process control 2-4
process creation 1-16
process data structures 1-14
process execution 1-16
process group ID 2-54, 2-128, 2-129
  set 2-128
process ID 2-54

process identification 2-4
process image
  new 2-34
process management 1-9
process priority
  automatic assignment 1-20
  change 2-88
process statistics 2-11
process suspension 2-94
process termination 2-40
process times
  child 2-165
  get 2-165
  parent 2-165
process-to-process communication 2-5
process trace 2-102
process user ID 2-129
processor
  difference, IBM Personal Computer AT and 032 Microprocessor D-7
processor user state 1-4
proc0 2-60
procl 2-60
profil system call 2-99
profile
  execution time 2-99
profile file 4-127
profile setting 4-127
profile, execution 3-248
profiler, operating system 6-106
program execution, DOS Services 3-79
programmable character set font 4-72.4
programmers workbench library 3-305
protocol modes 6-62
protocol procedure 3-376
pseudo-random number generator 3-317
pseudo-random numbers 3-118
pseudo-terminal device 6-107
psignal kernel subroutine C-24
ptrace system call 2-102
pty special file 6-107
publications
  related viii
push character back into input stream 3-369
putc kernel subroutine C-26
putc macro 3-309
putcb kernel subroutine C-27
putcf kernel subroutine C-27
putchar macro 3-309
putenv subroutine 3-310.1
putp subroutine 3-57
putpwent subroutine 3-312
puts subroutine 3-313
pututline subroutine 3-224
putw subroutine 3-309

Q

qconfig file 4-129
qdaemon to backend interaction B-2
qsort subroutine 3-315
query DMA 6-48
query hft device 3-352, 6-47
query physical device 6-41
query physical device identifiers 6-40
query presentation space 6-46
query terminal characteristics 3-352
queue
  message 2-111
  send message to 2-82
queue identifier 2-76
queue message
  read 2-79
  store 2-79
queued message, issue 3-259
queues, message C-24
queuing system B-1
quick sort 3-315
quit character 6-115

R

rand subroutine 3-317
random-number generator 3-317
random numbers 3-118
rasconf file 4-133
raw I/O C-3

raw subroutine 3-55, 3-158
read
  from a file, extended 2-106
  message from a queue 2-79
  open a file to 2-90
read a DOS file 3-98
read a file tree 3-200
read a password 3-217
read attribute file stanza 3-31
read from a file 2-106
read routine (ddread) C-10
read system call 2-106
read/write file pointer
  move 2-67, 3-104
readx system call 2-106
realloc subroutine 3-236
reboot system call 2-109
rebuild AIX kernel 3-21
rebuild kernel C-51
receive
  extended message from queue 2-85
refresh subroutine 3-55, 3-158
regcmp subroutine 3-318
regex subroutine 3-318
regexp facility 3-321
registers 1-4
  virtual 1-4
regular expression 3-318, 3-321
  advance 3-321
  compile 3-318, 3-321
  match 3-318
  step 3-321
related publications viii
relative path 1-32
release
  blocked signals 2-150
relocation, a.out 4-9
remainder function 3-167
remote hft 6-54
remote login 6-107
remove
  directory entry 2-174
remove a DOS Services directory 3-102
remove protocol procedure 3-376
rename a DOS file 3-100
rename system call 2-110.1
reopen all files 3-112
replace mode 6-69
report CPU time used 3-38
reposition the file pointer of a stream 3-196
resetterm subroutine 3-55
resetty subroutine 3-55, 3-158
resolution
  path name 1-30
retrieve a message, insert, or help text 3-263
return login name of user 3-233
return node ID 3-120.1
return node nickname 3-120.1
return status 1-37
rewind subroutine 3-196
rewrite existing file 2-27
ring, screen manager 6-50
rmdir system call 2-110.4
rmsgrcv kernel subroutine C-24.1
rmsgsnd kernel subroutine C-24
root directory
  change 2-23
routine libraries
  See libraries
routines
  See kernel subroutines
  See subroutines
RT PC hardware access D-7

S

saveterm subroutine 3-55
savetty subroutine 3-55, 3-158
sbrk system call 2-14
scanf subroutine 3-325
scanw subroutine 3-55, 3-158
SCCS delta table format 4-136
SCCS file format 4-135
sccsfile 4-135
schedule alarm 2-13
screen handling package 3-51
screen manager ring 6-50
screen optimization package 3-51
scroll subroutine 3-55, 3-158
scrollok subroutine 3-55, 3-159

search and update, linear 3-234
search trees, binary 3-364
search, binary 3-11
second-level interrupt handler C-9, C-18
seed48 subroutine 3-118
segment
  data 1-12
  stack 1-13
  text 1-12
sel_attr subroutine 3-159
select routine (ddselect) C-11
select support 6-12, 6-28, 6-108, 6-125
select system call 2-111
selwakeup kernel subroutine C-12
semaphores 2-115, 2-119, 2-122
semctl system call 2-115
semget system call 2-119
semop system call 2-122
send
  message to message queue 2-82
  signal to a process 2-60
  signal to process group 2-60
send a message to a queue 2-82
set-group-ID mode bit 2-36
set time 2-162
set-user-ID mode bit 2-36
set_term subroutine 3-55
setbuf subroutine 3-330
setgid system call 2-129
setgrent subroutine 3-210
setgroups system call 2-126
setjmp subroutine 3-332
setmpx kernel subroutine C-20
setpgrp system call 2-128
setpwent subroutine 3-219
setscrreg subroutine 3-55
setterm subroutine 3-56, 3-159
setting environment 4-127
setting the profile 4-127
setuid system call 2-129
setup_attr subroutine 3-159
setupterm subroutine 3-58
setutent subroutine 3-224
setvbuf subroutine 3-330
sgetl subroutine 3-334
shared memory
  control operations 2-135
shared memory segment
  attach 2-131
  detach 2-138
  get 2-140
shell command, issue 3-350
shell environment 2-35
shell variable 2-35
shell variable, value of 3-208
shift, single 5-9
shmat system call 2-131
shmctl system call 2-135
shmdt system call 2-138
shmget system call 2-140
shmop system calls 2-131, 2-135, 2-138, 2-140
shorten a file 2-50
SIGAIO signal 2-146
SIGALRM signal 2-146
sigblock system call 2-143
SIGBUS signal 2-145
SIGCLD signal 2-40, 2-41, 2-146, 2-148, 2-182, 3-262
SIGDANGER signal 2-145, 2-148
SIGFPE signal 2-145, 2-148, 2-157, 3-188
SIGGRANT signal 2-146
SIGHUP signal 2-41, 2-145
SIGILL signal 2-145
SIGINT signal 2-145
SIGIOINT signal 2-146
SIGIOT signal 2-145
SIGKILL signal 2-37, 2-143, 2-145
SIGMSG signal 2-146
signal
  block delivery 2-143
signal action 2-145
signal-catching function 2-145
signal facilities
  enhanced 2-156
signal handler 2-145
signal mask
  setting 2-152
signal overview 2-4
signal stack context 2-154
signal system call 2-145
signals 1-20, 2-145, 2-150
  release blocked 2-150

signals, device driver C-24
signals, floating-point 3-188
signals, software 3-340
sigpause system call 2-150
SIGPIPE signal 2-145
SIGPTY signal 2-146
SIGPWR signal 2-146, 2-148
SIGQUIT signal 2-145
SIGRETRACT signal 2-146
SIGSEGV signal 2-145
sigsetmask system call 2-152
SIGSOUND signal 2-146
sigstack system call 2-154
SIGSYS signal 2-145
SIGTERM signal 2-146
SIGTRAP signal 2-145
SIGUSR1 signal 2-146
SIGUSR2 signal 2-146
sigvec system call 2-156
sin subroutine 3-335
sine function 3-335
single-byte controls 5-11
single-shift control 5-9
single-user mode 4-3
sinh subroutine 3-337
sleep kernel subroutine C-21
sleep subroutine 3-338
SLIH C-9, C-18
software
  enhanced signal facilities 2-156
software signals 3-340
sort, quick 3-315
sound data 6-66
space
  allocation change for data segment 2-14
space subroutine 3-296
special character definitions for eqn and neqn 5-54
special file 2-71
  create 2-69
special files 1-24, 1-40
specification of text file format 4-82
splhi kernel subroutine C-19
splx kernel subroutine C-19
spl0-spl7 kernel subroutines C-19
sprintf subroutine 3-300
sputl subroutine 3-334
sqrt subroutine 3-128
square root 3-128
srand subroutine 3-317
srand48 subroutine 3-118
sscanf subroutine 3-325
ssignal subroutine 3-340
SS1—SS4 5-9
stack
  signal 2-154
stack segment 1-13
standard I/O 3-309
standard I/O subroutine library 3-342
standard interprocess communication package 3-198
standend subroutine 3-56, 3-160
standout subroutine 3-56, 3-160
stanza, add 3-23
stanza, delete 3-27
stanza, read 3-31
stanza, replace 3-23
stanza, write 3-23
start
  character 6-115
  system 2-109
  virtual machine 2-58
stat structure 5-69
stat system call 2-159
stat.h header file 5-69
state of processor
  operating system 1-4
  user 1-4
statistics
  file system 2-178
  process 2-11
statistics, file system 2-178
status
  check I/O 2-111
  file 2-159
status of a DOS file, get 3-106
status of DOS Services device, get 3-114
status, stream 3-165
statusfile parameter B-2
stdio subroutine library 3-342
stdipc (ftok subroutine) 3-198
stime system call 2-162

stop
  wait for child process to 2-182
stop character 6-115
storage image file 4-39
store
  message from a queue 2-79
store subroutine 3-63
strategy routine (ddstrategy) C-17
strcat subroutine 3-344
strchr subroutine 3-344
strcmp subroutine 3-344
strcpy subroutine 3-344
strcspn subroutine 3-344
stream closing and flushing 3-163
stream I/O 3-309
stream open 3-168
stream status 3-165
stream, assigning buffering to 3-330
stream, data
  3270 6-11
stream, get character or word from 3-204
string from a stream, obtaining 3-221
string handling 3-278
string operations 3-285, 3-344
  international character support 3-272
string to integer conversion 3-347
string, write to a stream 3-313
strlen subroutine 3-344
strncat subroutine 3-344
strncmp subroutine 3-344
strncpy subroutine 3-344
strpbrk subroutine 3-344
strrchr subroutine 3-344
strspn subroutine 3-344
strtod subroutine 3-8
strtok subroutine 3-344
strtol subroutine 3-347
structures
  a.out 4-5
  a.out relocation 4-9
  accounting file 4-15
  archive file member 4-18
  backup 4-23
  cpio 4-41
  Define_Code SVC 6-7
  device driver config 6-9
  devinfo 4-57, 6-100
  directory file 4-60
  disk config 6-8
  diskette customizing 6-17
  fd devinfo 6-18
  fullstat 5-56.2
  HD devinfo 6-21
  inode 4-92
  ipc_perm 2-6
  lprio.h 6-100
  lprmode 6-100
  LPRUDE 6-101
  minidisk customize 6-20
  mnttab.h 4-108
  oprmode 6-101
  portstatus 4-122
  process data 1-14
  stat 5-69
  superblock 4-75
  symbol table 4-10
  tacct.h 4-16
  tape archive header 4-146
  termio 6-116
  VRM 1-6
  VRM Query_Device call 6-9
structures, file
  See file formats
sublevel, interrupt C-18
subroutine libraries
  See libraries
subroutines
  See also kernel subroutines
  del_ipc_prof 3-64.1
  find_ipc_prof 3-166.1
subsystem
  buffer 1-36
  file i/o 1-36
subwin subroutine 3-56, 3-160
subyte kernel subroutine C-13
superblock 1-25
  update 2-163
superbox subroutine 3-160
supervisor call instruction 1-4
supervisor calls, AIX
  See system calls
suspend

process 2-94
suspend execution 3-338
suword kernel subroutine C-13
SVCs, AIX
See system calls
swab subroutine 3-349
swap bytes 3-349
switch table, device C-3
symbol table structure 4-10
symbols, display 5-24
sync system call 2-163
synchronize a DOS file 3-83
syntax v
sys_errlist 3-294
sys_nerr 3-294
system
reboot 2-109
system calls
difference from subroutines 2-2
errno values A-1
functional summary 2-2
system data types, primitive 5-75
system error messages 3-294
system file 4-139
system name
extended 2-172
get 2-172
system overview 1-3
system profiler 6-106
system subroutine 3-350
system volume format 4-74

T

TABDLY 6-118
table
call switch 1-36
device switch 1-36
DOS function call D-2
table, mounted file system 4-108
tabs, non-standard 4-82
TAB0 6-118
TAB1 6-118
TAB2 6-118
TAB3 6-118
tacct.h structure 4-16
tail, of screen manager ring 6-50
tan subroutine 3-335
tangent function 3-335
tanh subroutine 3-337
tape archive header structure 4-146
tape driver file 6-111
tape special file 6-111
tar file 4-146
tcflsh 6-126
tcgeta 6-125
tcsbrk 6-126
tcseta 6-125
tcsetaf 6-125
tcsetaw 6-125
tcxonc 6-126
tdelete subroutine 3-364
tempnam subroutine 3-355
temporary file creation 3-354
temporary file creation, DOS 3-92
temporary file naming 3-355
TERM environment variable 5-72
TERM variable 3-352
termcap
emulation using terminfo 3-59
termdef subroutine 3-352
terminal capability data base 4-148
terminal characteristics 3-352
terminal file name generation 3-44
terminal interface control 6-131
terminal name 3-367
terminal, data base 4-148
terminate
wait for child process to 2-182
terminate a process 2-40
terminfo file 4-148
termio file 6-114
termio structures 6-116
text
lock 2-97
unlock 2-97
text file format specification 4-82
text segment 1-12
text, help, issue 3-252
tgetent subroutine 3-59

tgetflag subroutine 3-59
tgetnum subroutine 3-59
tgetstr subroutine 3-59
tgoto subroutine 3-59
thresholds, locator 6-64
tilde accent character 5-10
time
  get 2-164
  set 2-162
time format 3-288
time profile
  execution time 2-99
time structure 3-291
time system call 2-164
time to string conversion 3-46
time used report, CPU 3-38
time_t data type 5-75
timeout kernel subroutine C-21
times system call 2-165
timezone external variable 3-46
tmpfile subroutine 3-354
tmpnam subroutine 3-355
toascii subroutine 3-39
tolower subroutine 3-39
_tolower subroutine 3-39
touchwin subroutine 3-56, 3-161
toupper subroutine 3-39
_toupper subroutine 3-39
tparm subroutine 3-58
tputs subroutine 3-58, 3-59
trace 3-357
  child process execution 2-102
trace channel, check whether enabled 3-357
trace collector, AIX 3-362
trace daemon 3-359
trace driver 6-128
trace special file 6-128
trace, device driver C-32
trace, kernel C-32
trace_on subroutine 3-357
traceoff subroutine 3-56
traceon subroutine 3-56
trackloc subroutine 3-161
trailer record 4-42
translate
  characters 3-39, 3-276
  group IDs 2-21
  owner IDs 2-21
trap, floating-point exception 3-188
trap, kernel 1-35
trc_start subroutine 3-359
trc_stop subroutine 3-359
trcunix subroutine 3-362
tree, read 3-200
trees, binary search 3-364
trigonometric functions 3-335
trsave kernel subroutine C-32
tsearch subroutine 3-364
tstp subroutine 3-161
tty special file 6-131
ttyname subroutine 3-367
twalk subroutine 3-364
two-byte characters 5-9
typeahead subroutine 3-56
types.h header file 5-75
tzname external variable 3-46
tzset subroutine 3-46

U

U.S. English keyboard 6-79
uint data type 5-75
ulimit system call 2-167
ulong data type 5-75
umask system call 2-169
umlaut accent character 5-10
umount system call 2-170
uname system call 2-172
unamex system call 2-172
unctrl subroutine 3-56, 3-162
ungetc subroutine 3-369
Unix error collector 3-126
unlink a DOS file 3-110
unlink system call 2-174
unlock
  region of a file 2-64
unlock a region of a DOS file 3-88
unmount
  file system 2-170
untimeout kernel subroutine C-22

update
  delayed blocks 2-163
  i-nodes 2-163
  superblock 2-163
update, linear 3-234
user ID
  get 2-55
  set 2-129
user information 2-176
user information name, find value 3-223
user limits 2-167
user login name 3-62
user login name, obtaining 3-233
user mode 1-10
user mode addressing 1-12
user name 3-62
ushort data type 5-75
usrchar kernel subroutine C-20
usrinfo system call 2-176
ustat system call 2-178
utime system call 2-180
utmp file 4-170
utmp file entry access 3-224
utmp file, find user's slot 3-368
utmpname subroutine 3-224
uvmount system call 2-180.3

V

value of environment variable 3-208
value of user information name, find 3-223
values.h header file 5-77
varargs argument list, print 3-374
varargs macro 3-371
variable-length parameter list 3-371, 3-374
vec_clear kernel subroutine C-18
vec_init kernel subroutine C-18
verify program assertion 3-7
verify, write 3-243
vfprint subroutine 3-374
vidattr subroutine 3-58
vidputs subroutine 3-58
virtual
  memory 1-10
  registers 1-4
  terminal data (VTD) 6-61
virtual machine
  characteristics 1-3
  control registers 1-4
  IPL 2-58
  restart 2-109
  start 2-58
  wait for termination 2-58
virtual memory image 6-103
vmount system call 2-180.5
vprintf subroutine 3-374
vrcppr subroutine 3-376
VRM device driver 1-36
VRM Query_Device call 6-9
VRM structure 1-6
vscroll subroutine 3-162
vsprint subroutine 3-374
VTD 6-61
VTDLY 6-118
VT0 6-118
VT1 6-118

W

waddch subroutine 3-52, 3-134
waddfld subroutine 3-135
waddstr subroutine 3-52, 3-135
wait
  for I/O activity 2-111
  for signal 2-94
  virtual machine 2-58
wait system call 2-182
waitvm system call 2-58
wakeup kernel subroutine C-21
walk a file tree 3-200
wattroff subroutine 3-56
wattron subroutine 3-56
wattrset subroutine 3-56
wchgat subroutine 3-136
wclear subroutine 3-56, 3-137
wclrtobot subroutine 3-56, 3-137
wclrtoeol subroutine 3-56, 3-137
wcolorend subroutine 3-137

wcolorout subroutine 3-138
wdelch subroutine 3-56, 3-139
wdeleteln subroutine 3-56, 3-139
werase subroutine 3-56, 3-150
wgetch subroutine 3-56, 3-151
wgetstr subroutine 3-56, 3-152
winch subroutine 3-56, 3-152
winsch subroutine 3-56, 3-153
winsertln subroutine 3-56, 3-153
wmove subroutine 3-56, 3-154
wnoutrefresh subroutine 3-56
word, get from stream 3-204
workbench library 3-305
wprintw subroutine 3-56, 3-157
wrefresh subroutine 3-56, 3-158
write
  file to permanent storage 2-48
  open a file to 2-90
  to a file 2-184
write a string to a stream 3-313
write characters 3-309
write password file entry 3-312
write routine (ddwrite) C-10
write stanza 3-23
write system call 2-184
write to a DOS file 3-116
write to a stream 3-309
write-verify 3-243
write words 3-309
writex system call 2-184
writing a helper program C-50
wscanw subroutine 3-57, 3-158
wsetscrreg subroutine 3-57
wstandend subroutine 3-57, 3-160
wstandout subroutine 3-57, 3-160
wtmp file 4-170

X

XCASE 6-120

Y

y0, y1, yn subroutines 3-9

Z

zombie process 2-40

Numerics

3-byte integer conversion to long integers 3-232
3270 data stream 6-11
3270 device driver 6-11
3278/79 Emulation Adapter 6-11
68881 floating-point processor 3-170, 3-183, 3-190

The IBM RT PC
Family

Reader's Comment Form

IBM RT PC AIX Operating System Technical Reference SC23-0808-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding set up, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

IBM RT PC AIX Operating System Technical Reference ____________ **SC23-0808-0**

Book Title **Order No.**

Book Evaluation Form

Your comments can help us produce better books. You may use this form to communicate your comments about this book, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you. Please take a few minutes to evaluate this book as soon as you become familiar with it. Circle Y (Yes) or N (No) for each question that applies and give us any information that may improve this book.

Y N Is the purpose of this book clear?

Y N Is the table of contents helpful?

Y N Is the index complete?

Y N Are the chapter titles and other headings meaningful?

Y N Is the information organized appropriately?

Y N Is the information accurate?

Y N Is the information complete?

Y N Is only necessary information included?

Y N Does the book refer you to the appropriate places for more information?

Y N Are terms defined clearly?

Y N Are terms used consistently?

Y N Are the abbreviations and acronyms understandable?

Y N Are the examples clear?

Y N Are examples provided where they are needed?

Y N Are the illustrations clear?

Y N Is the format of the book (shape, size, color) effective?

Other Comments

What could we do to make this book or the entire set of books for this system easier to use?

Optional Information

Your name ____________

Company name ____________

Street address ____________

City, State, ZIP ____________

No postage necessary if mailed in the U.S.A.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 997, Building 998
11400 Burnet Rd.
Austin, Texas 78758

Fold and tape

Fold and tape

Cut or Fold Along Line

Tape

Please Do Not Staple

Tape

SN20-9855-00

79X3872