900 Series HP 3000 Computer Systems

MPE/iX Developer's Kit Reference Manual Volume I

HEWLETT PACKARD

HP Part No. 36430-90001
Printed in U.S.A. 1994

Second Edition
E0494

FINAL TRIM SIZE : 7.0 in x 8.5 in

FINAL TRIM SIZE : 7.0 in x 8.5 in

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time that this document was issued. Many product releases do not require changes to the document; therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	**Date**	**Software Version**
First Edition	October 1992	A.00.00
Second Edition	April 1994	C.50.00

Preface

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s not based on PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

The *MPE/iX Developer's Kit Reference Manual, Volume 1* (36430-90001) describes the POSIX/iX library provided with the MPE/iX Developer's Kit (36430A) on 900 Series HP 3000 computer systems. This manual is intended for experienced C programmers.

This manual is organized as follows:

Chapter 1 **Introduction** provides a summary overview of the POSIX/iX library.

Chapter 2 **Using the POSIX/iX Library** provides information on general C library considerations and how to develop applications using the MPE/iX Shell and Utilities.

Chapter 3 **POSIX/iX Library Implementation Considerations** describe important MPE/iX implementation details you need to know when using POSIX/iX library functions.

Chapter 4 **POSIX/iX Library Function Descriptions** presents the syntax and descriptions of POSIX/iX library functions, arranged alphabetically.

Chapter 5 **POSIX/iX Header Descriptions** describes the contents of header files required by the POSIX/iX library.

Conventions

`nonitalics` Within syntax descriptions, nonitalicized words represent literals. Enter them exactly as shown. This includes angle brackets appearing within syntactic descriptions. For example,

```
#include <unistd.h>
```

Nonitalicized words and punctuation characters appear in `computer font`. In the following example, you must provide the keyword, function name, parentheses, and trailing semicolon:

```
int ccode();
```

italics Within syntax descriptions, italicized words denote argument names, program names, or strings that you must replace with an appropriate value. In the following example, you must replace *number* and *denom* with the respective integers you want to pass to the `div` function:

`div(`*number*`, `*denom*`);`

[ ] Within syntax descriptions, italicized brackets surround optional elements. For example, the *item* list in the `scanf()` function call is optional:

`scanf(`*format [, item [, item]. . .]* `);`

. . . Within syntax descriptions, a horizontal ellipses indicates that a previous element can be repeated. For example:

[, item]. . .

Within examples, vertical and horizontal ellipses may show where portions of the example were omitted.

Contents

Tables

1

Introduction

This chapter provides a summary overview of the POSIX/iX library available through the MPE/iX Developer's Kit (product # 36430A). The following topics are discussed in this chapter:

- What is the POSIX/iX library?
- How to use this manual.
- Overview of the POSIX standards.
- Understanding MPE/iX.
- Summary of POSIX/iX library functions.

What Is the POSIX/iX Library?

The POSIX/iX library is an implementation on 900 Series HP 3000 computer systems of many of the C library functions and features defined in the following:

- IEEE Standard 1003.1-1990 (ISO/IEC 9945-1:1990)
- Appendix B of the IEEE P1003.2/D11.2

The POSIX Standards

The Institute of Electrical and Electronics Engineers (IEEE) has been investigating the application of standards to information technology. This work has led to a set of standards known collectively as IEEE 1003, or Portable Operating System Interface (POSIX). The IEEE 1003 is actually a group of individual standards that address specific areas of information technology.

This book, *MPE/iX Developer's Kit Reference Manual Volume 1* (36430-90001) and *MPE/iX Developer's Kit Reference Manual Volume 2* (36430-90002), describes the MPE/iX implementation of the IEEE 1003 standards:

1003.1	C language bindings
1003.2	Shell commands and utilities

The POSIX/iX library is implemented according to the standards set forth in the 1990 revision of the POSIX.1 standard.

POSIX.1 is concerned with C language application programming interfaces (APIs) and contains over 200 function calls (defined by POSIX and ANSI), along with type definitions, header files, and a data interchange format. POSIX.1 allows C applications a standard programmatic interface to make system calls, I/O requests, and general library calls.

The POSIX.1 standard is defined in the book *Information Technology - Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]; ISO/IEC 9945-1:1990;*; IEEE, 1990; ISBN 1-55937-061-0.

How to Use This Manual

This manual is intended to be used with the following four manuals:

- *HP C/iX Library Reference Manual* (30026-90001)
- *MPE/iX Developer's Kit Reference Manual Volume 2* (36430-90002)
- *MPE/iX Shell and Utilities Reference Manual, Volumes 1 and 2* (36431-60001)
- *The POSIX.1 Standard - A Programmer's Guide* (36430-90003)

This manual describes C library functions defined by the POSIX.1 standard. Additional C library functions available through the POSIX/iX library are documented in the *HP C/iX Library Reference Manual* (30026-90001).

Using the HP C/iX Library Reference Manual

The *HP C/iX Library Reference Manual* (30026-90001) contains reference descriptions of ANSI C library functions that are required by the POSIX.1 standard. Some of the functions in the *HP C/iX Library Reference Manual* (30026-90001):

- Do not correctly describe the POSIX/iX function. Refer to this manual, the *MPE/iX Developer's Kit Reference Manual Volume 1* (36430-90001) for the appropriate descriptions. The following is a list of those functions:
  - `access()`
  - `close()`
  - `creat()`
  - `dup()`
  - `getenv()`
  - `getopt()`
  - `getpid()`
  - `isatty()`
  - `lseek()`
  - `open()`
  - `read()`
  - `sleep()`
  - `system()`
  - `time()`
  - `write()`
- Are not implemented in the POSIX/iX library. At this time there is only one function that is not implemented:
  - `link()`
- Support the HP C/iX compiler product and do not apply to the MPE/iX product. Table 1-1 lists the POSIX/iX functions, which functions are implemented in POSIX/iX, and where you can find the POSIX/iX function description.

The POSIX.1 standard requires that certain ANSI C library functions have additional capabilities or characteristics. These extensions to the behavior of ANSI C library functions are described in the *HP C/iX Library Reference Manual* (30026-90001) under the appropriate function description. This manual is available as a special core supplement to the MPE/iX FOS manual set.

Using the MPE/iX Developer's Kit Reference Manual, Volume 2

The *MPE/iX Developer's Kit Reference Manual Volume 2* (36430-90002) contains descriptions of C library functions available through additional relocatable libraries provided with the MPE/iX Developer's Kit. Refer to the *MPE/iX Developer's Kit Reference Manual Volume 2* (36430-90002) for more information.

Using the MPE/iX Shell and Utilities Reference Manual

The *MPE/iX Shell and Utilities Reference Manual, Volumes 1 and 2* (36431-60001) provide descriptions of shell commands, headers, and utilities defined in the POSIX.1 standard or in Appendix B of IEEE P1003.2/D11.2. For more information about this, refer to the *MPE/iX Shell and Utilities Reference Manual, Volumes 1 and 2* (36431-60001). This manual set is available as part of the Developer's Kit.

Using The POSIX.1 Standard - A Programmer's Guide

The POSIX.1 Standard - A Programmer's Guide (36430-90003) provides additional information on the POSIX.1 standard and POSIX.1 application programming.

The POSIX.1 Standard - A Programmer's Guide (36430-90003) is written to describe how to program in a strictly conforming POSIX.1 environment. You should understand all implementation considerations associated with the POSIX/iX library before using *The POSIX.1 Standard - A Programmer's Guide* (36430-90003) as a programming aid for application development on MPE/iX.

Understanding MPE/iX

The MPE/iX Developer's Kit provides facilities that allow you to develop portable applications while minimizing the need to understand underlying MPE/iX operating system features. However, because the current implementation of the POSIX/iX library does not conform to the POSIX.1 standard, some of the topics discussed in this manual require that you have an understanding of underlying features of the MPE/iX operating system.

Additional MPE/iX documentation is available that contains detailed information about MPE/iX features not discussed in detail in this manual. This manual briefly summarizes these features and provides pointers to the manuals where you can acquire additional information.

The following manual provides an introduction to many of the MPE/iX features that you'll need to understand:

- *New Features of MPE/iX: Using the Hierarchical File System* (32650-90351)

These manuals are available as a special core supplement to the MPE/iX FOS manual set.

Summary of POSIX/iX Library Functions

The following table lists the functions available through the POSIX/iX library. It also indicates the manual in which function descriptions are located.

Table 1-1. Summary of POSIX/iX Library Functions

Function Name	Standards Definition	Description Location
`a64l()`		*HP C/iX Library Reference Manual*
`abort()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`abs()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`access()`	POSIX.1	This manual
`acos()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`alarm()`	POSIX.1	This manual
`asctime()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`asin()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`assert()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`atan()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`atan2()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`atexit()`	ANSI C	*HP C/iX Library Reference Manual*
`atof()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`atoi()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`atol()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
Bessel functions		*HP C/iX Library Reference Manual*
`brk()`		*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`bsearch()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`calloc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`catread()`		*HP C/iX Library Reference Manual*
`ccode()`		*HP C/iX Library Reference Manual*
`ceil()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`cfgetispeed()`	POSIX.1	Not currently implemented
`cfgetospeed()`	POSIX.1	Not currently implemented
`cfsetispeed()`	POSIX.1	Not currently implemented
`cfsetospeed()`	POSIX.1	Not currently implemented
`chdir()`	POSIX.1	This manual
`chmod()`	POSIX.1	This manual
`chown()`	POSIX.1	This manual
`clc05()`		*HP C/iX Library Reference Manual*
`clearerr()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`clock()`	ANSI C	*HP C/iX Library Reference Manual*
`close()`	POSIX.1	This manual
`closedir()`	POSIX.1	This manual
`cos()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`confstr()`	POSIX.2	This manual
`cosh()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`creat()`	POSIX.1	This manual

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`crypt()`		*HP C/iX Library Reference Manual*
`ctermid()`	POSIX.1	This manual
`ctime()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`difftime()`	ANSI C	*HP C/iX Library Reference Manual*
`div()`	ANSI C	*HP C/iX Library Reference Manual*
`dup()`	POSIX.1	This manual
`dup2()`	POSIX.1	This manual
`ecvt()`		*HP C/iX Library Reference Manual*
`encrypt()`		*HP C/iX Library Reference Manual*
`erf()`		*HP C/iX Library Reference Manual*
`erfc()`		*HP C/iX Library Reference Manual*
`execl()`	POSIX.1	This manual
`execle()`	POSIX.1	This manual
`execlp()`	POSIX.1	This manual
`execv()`	POSIX.1	This manual
`execve()`	POSIX.1	This manual
`execvp()`	POSIX.1	This manual
`exit()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`_exit()`	POSIX.1	This manual
`exp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fabs()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fclose()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`fcntl()`	POSIX.1	This manual
`fcvt()`		*HP C/iX Library Reference Manual*
`fdopen()`	POSIX.1	*HP C/iX Library Reference Manual*
`feof()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`ferror()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fflush()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fgetc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fgetpos()`	ANSI C	*HP C/iX Library Reference Manual*
`fgets()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fileno()`	POSIX.1	*HP C/iX Library Reference Manual*
`floor()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fmod()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fnmatch()`	POSIX.2	This manual
`fopen()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fork()`	POSIX.1	This manual
`fpathconf()`	POSIX.2	This manual
`fprintf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fprintmsg()`		*HP C/iX Library Reference Manual*
`fputc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fputs()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fread()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`free()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`freopen()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`frexp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fscanf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fseek()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fsetpos()`	ANSI C	*HP C/iX Library Reference Manual*
`fstat()`	POSIX.1	This manual
`ftell()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`fwrite()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`gamma()`		*HP C/iX Library Reference Manual*
`gcvt()`		*HP C/iX Library Reference Manual*
`getc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`getchar()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`getcwd()`	POSIX.1	This manual
`getegid()`	POSIX.1	This manual
`getenv()`	POSIX.1, ANSI C	This manual
`geteuid()`	POSIX.1	This manual
`getgid()`	POSIX.1	This manual
`getgrgid()`	POSIX.1	This manual
`getgrnam()`	POSIX.1	This manual
`getgroups()`	POSIX.1	This manual
`getlogin()`	POSIX.1	This manual

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`getmsg()`		*HP C/iX Library Reference Manual*
`getopt()`	IEEE, POSIX.2	This manual
`getpgrp()`	POSIX.1	This manual
`getpid()`	POSIX.1	This manual
`getppid()`	POSIX.1	This manual
`getpwnam()`	POSIX.1	This manual
`getpwuid()`	POSIX.1	This manual
`gets()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`getuid()`	POSIX.1	This manual
`getw`		*HP C/iX Library Reference Manual*
`glob()`	POSIX.2	This manual
`globfree()`	POSIX.2	This manual
`gmtime()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`hcreate()`		*HP C/iX Library Reference Manual*
`hdestroy()`		*HP C/iX Library Reference Manual*
`hsearch()`		*HP C/iX Library Reference Manual*
`hypot()`		*HP C/iX Library Reference Manual*
`ioctl()`		This manual
`isalnum()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isalpha()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isatty()`	POSIX.1	This manual
`iscntrl()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`isdigit()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isgraph()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`islower()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isprint()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`ispunct()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isspace()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isupper()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`isxdigit()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`kill()`	POSIX.1	This manual
`l3tol()`		*HP C/iX Library Reference Manual*
`l64a()`		*HP C/iX Library Reference Manual*
`labs()`	ANSI C	*HP C/iX Library Reference Manual*
`ldexp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`ldiv()`	ANSI C	*HP C/iX Library Reference Manual*
`lfind()`		*HP C/iX Library Reference Manual*
`link()`	POSIX.1	Not currently implemented
`localeconv()`	ANSI C	*HP C/iX Library Reference Manual*
`localtime()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`log()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`log10()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`longjmp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`lsearch()`		*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`lseek()`	POSIX.1	This manual
`ltol3()`		*HP C/iX Library Reference Manual*
`mallinfo()`		*HP C/iX Library Reference Manual*
`malloc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`mallopt()`		*HP C/iX Library Reference Manual*
`matherr()`		*HP C/iX Library Reference Manual*
`mblen()`	ANSI C	*HP C/iX Library Reference Manual*
`mbstowcs()`	ANSI C	*HP C/iX Library Reference Manual*
`mbtowc()`	ANSI C	*HP C/iX Library Reference Manual*
`memccpy()`		*HP C/iX Library Reference Manual*
`memchr()`	ANSI C	*HP C/iX Library Reference Manual*
`memcmp()`	ANSI C	*HP C/iX Library Reference Manual*
`memcpy()`	ANSI C	*HP C/iX Library Reference Manual*
`memmove()`	ANSI C	*HP C/iX Library Reference Manual*
`memset()`	ANSI C	*HP C/iX Library Reference Manual*
`mkdir()`	POSIX.1	This manual
`mkfifo()`	POSIX.1	This manual
`mknod()`		This manual
`mktemp()`		*HP C/iX Library Reference Manual*
`mktime()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`modf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`_mpe_fileno()`		*HP C/iX Library Reference Manual*
`offsetof()`	ANSI C	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`open()`	POSIX.1	This manual
`opendir()`	POSIX.1	This manual
`pathconf()`	POSIX.2	This manual
`pause()`	POSIX.1	This manual
`pclose()`	POSIX.2	This manual
`perror()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`pipe()`	POSIX.1	This manual
`popen()`	POSIX.2	This manual
`pow()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`printf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`printmsg()`		*HP C/iX Library Reference Manual*
`putc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`putchar()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`putenv()`		This manual
`puts()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`putw()`		*HP C/iX Library Reference Manual*
`qsort()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`raise()`	ANSI C	*HP C/iX Library Reference Manual*
`rand()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`rand48()`		*HP C/iX Library Reference Manual*
`read()`	POSIX.1	This manual
`readdir()`	POSIX.1	This manual

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`realloc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`regcomp()`	POSIX.2	This manual
`regerror()`	POSIX.2	This manual
`regexec()`	POSIX.2	This manual
`regfree()`	POSIX.2	This manual
`remove()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`rename()`	POSIX.1, ANSI C	This manual
`rewind()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`rewinddir()`	POSIX.1	This manual
`rmdir()`	POSIX.1	This manual
`sbrk()`		*HP C/iX Library Reference Manual*
`scanf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`setbuf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`setgid()`	POSIX.1	Not currently implemented
`setjmp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`setkey()`		*HP C/iX Library Reference Manual*
`setlocale()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`setpgid()`	POSIX.1	Not currently implemented
`setsid()`	POSIX.1	Not currently implemented

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`setuid()`	POSIX.1	This manual
`setvbuf()`	ANSI C	*HP C/iX Library Reference Manual*
`sigaction()`	POSIX.1	This manual
`sigaddset()`	POSIX.1	This manual
`sigdelset()`	POSIX.1	This manual
`sigemptyset()`	POSIX.1	This manual
`sigfillset()`	POSIX.1	This manual
`sigismember()`	POSIX.1	This manual
`siglongjmp()`	POSIX.1	This manual
`signal()`	ANSI C	*HP C/iX Library Reference Manual*
`sigpending()`	POSIX.1	This manual
`sigprocmask()`	POSIX.1	This manual
`sigsetjmp()`	POSIX.1	This manual
`sigsuspend()`	POSIX.1	This manual
`sin()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`sinh()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`sleep()`	POSIX.1	This manual
`sprintf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`sprintmsg()`		*HP C/iX Library Reference Manual*
`sqrt()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`srand()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`sscanf()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`stat()`	POSIX.1	This manual
`strcat()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strchr()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strcmp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strcoll()`	ASNI C	*HP C/iX Library Reference Manual*
`strcpy()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strcspn()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strerror()`	ANSI C	*HP C/iX Library Reference Manual*
`strftime()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strlen()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strncat()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strncmp()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strncpy()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strpbrk()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strrchr()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strspn()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strstr()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strtod()`	ANSI C	*HP C/iX Library Reference Manual*
`strtok()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`strtol()`	ANSI C	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
strtoul()	ANSI C	*HP C/iX Library Reference Manual*
strxfrm()	ANSI C	*HP C/iX Library Reference Manual*
swab()		*HP C/iX Library Reference Manual*
symlink()|	This manual	
sysconf()	POSIX.2	This manual
system()	ANSI C, POSIX.2	This manual
symlink()		This manual
tan()	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
tanh()	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
tcdrain()	POSIX.1	Not currently implemented
tcflow()	POSIX.1	Not currently implemented
tcflush()	POSIX.1	Not currently implemented
tcgetattr()	POSIX.1	Not currently implemented
tcgetpgrp()	POSIX.1	Not currently implemented
tcsendbreak()	POSIX.1	Not currently implemented
tcsetattr()	POSIX.1	Not currently implemented
tcsetpgrp()	POSIX.1	Not currently implemented
tdelete()		*HP C/iX Library Reference Manual*
tfind()		*HP C/iX Library Reference Manual*
time()	POSIX.1, ANSI C	This manual
times()	POSIX.1	This manual

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`tmpfile()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`tmpnam()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`toascii()`		*HP C/iX Library Reference Manual*
`tolower()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`toupper()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`tsearch()`		*HP C/iX Library Reference Manual*
`ttyname()`	POSIX.1	This manual
`twalk()`		*HP C/iX Library Reference Manual*
`tzset()`	POSIX.1	*HP C/iX Library Reference Manual*
`umask()`	POSIX.1	This manual
`uname()`	POSIX.1	This manual
`ungetc()`	ANSI C, POSIX.1	*HP C/iX Library Reference Manual*
`unlink()`	POSIX.1	This manual
`utime()`	POSIX.1	This manual
`va_arg()`	ANSI C	*HP C/iX Library Reference Manual*
`va_end()`	ANSI C	*HP C/iX Library Reference Manual*
`va_start()`	ANSI C	*HP C/iX Library Reference Manual*
`vfprintf()`	ANSI C	*HP C/iX Library Reference Manual*
`vprintf()`	ANSI C	*HP C/iX Library Reference Manual*

Table 1-1. Summary of POSIX/iX Library Functions (continued)

Function Name	Standards Definition	Description Location
`vsprintf()`	ANSI C	*HP C/iX Library Reference Manual*
`wait()`	POSIX.1	This manual
`waitpid()`	POSIX.1	This manual
`wcstombs()`	ANSI C	*HP C/iX Library Reference Manual*
`wctomb()`	ANSI C	*HP C/iX Library Reference Manual*
`wordexp()`	POSIX.2	This manual
`wordfree()`	POSIX.2	This manual
`write()`	POSIX.1	This manual

2

Using the POSIX/iX Library

The POSIX/iX library provides an extensive library of C functions. The functions provide facilities for such operations as input, output, process management, signal management, mathematics, string manipulation, and time and date operations.

This chapter provides information on the following subjects:

- how the POSIX/iX library is organized
- specifying the `_POSIX_SOURCE` macro
- how to develop applications using the MPE/iX Shell and Utilities
- general input/output considerations
- ANSI C library functions that have modified behavior

Organization of the POSIX/iX Library

The POSIX/iX library consists of three relocatable library files:

- The file `/lib/libc.a` is a relocatable library file that includes C library functions defined by ANSI C, the POSIX.1 standard, and IEEE P1003.2/D11.2.
- The file `/lib/libm.a` is a relocatable library file containing all ANSI C math library functions.
- The file `/lib/libM.a` is a relocatable library file containing all common usage math library functions.

The POSIX/iX library

The POSIX/iX library file, `/lib/libc.a`, contains four classes of C library functions:

- functions defined by ANSI C
- functions defined by the POSIX.1 standard
- functions defined by IEEE P1003.2/D11.2
- functions provided to increase portability between MPE/iX and HP-UX

The POSIX/iX math library

The POSIX/iX math library consists of additional mathematical functions, such as trigonometric and logarithmic functions, that perform floating-point operations. These math library functions perform in a manner defined by ANSI C.

Note It is recommended that you use the POSIX/iX math library when developing applications using the MPE/iX Developer's Kit.

The common usage math library

The common usage math library consists of the same library functions available in the POSIX/iX math library; however, common usage math library functions perform in a pre-ANSI manner that does not conform to either ANSI C or the POSIX.1 standard.

The primary difference between the two math libraries is the manner in which errors are handled, such as attempting to compute the square root of a negative value. POSIX/iX math library behavior causes the library to call a user-written function named `_matherr` if one is provided, and no error message is displayed. Common usage math library behavior causes the library to call a user-written function named `matherr` if one is provided; otherwise, an error message is displayed.

Specifying the _POSIX_SOURCE Feature Test Macro

An application that includes a header described by the POSIX.1 standard must specify the `_POSIX_SOURCE` feature test macro prior to any instance of that header being included in the source file. When `_POSIX_SOURCE` is specified in the source file, the following conditions are true:

- All symbols required by the POSIX.1 standard are made visible to the application.
- Symbols that are explicitly permitted, but not required, by the POSIX.1 standard are made visible to the application.
- Additional symbols not required or explicitly permitted by the POSIX.1 standard are not made visible.

Input/Output Considerations

The POSIX/iX library provides two mechanisms to operate on MPE/iX files:

- streams
- file descriptors

Both streams and file descriptors serve as "handles" to the underlying file.

MPE/iX supported many different file types and file record formats; however, the POSIX/iX library supports operations only on files whose MPE/iX record format is byte-stream (referred to as byte-stream files). All files created or opened through POSIX/iX library functions are MPE/iX byte-stream files. Attempts to open an MPE/iX file whose type is other than byte-stream results in an error. This applies to emulators.

The *HP C/iX Library Reference Manual* (30026-90001) describes the behavior of C library functions when operating on various MPE/iX file types. You should ignore these references and pay attention only to those sections that describe the behavior of library functions when they are operating on MPE/iX byte-stream files.

Streams

Streams are abstractions over file descriptors in order to provide buffered I/O defined by ANSI C. ANSI C defines two types of streams, the text stream and the binary stream; however, the POSIX/iX library does not distinguish between text and binary streams. For more information about streams, refer to the *HP C/iX Library Reference Manual* (30026-90001).

Note The term "stream" should not be confused with the MPE/iX file whose record format is "byte-stream".

File descriptors

The POSIX.1 standard defines an additional method of accessing a file, through the use of file descriptors. A file descriptor is a per-process nonnegative integer used to identify an open file. For example, when creating or opening a file using the `open()` function, a file descriptor whose type is integer is associated with the underlying file description and returned to the calling process. All subsequent accesses of that file are performed through the file descriptor. The buffered I/O performed for streams is not performed when accessing a file through its file descriptor.

Extended Behavior of ANSI C Library Functions

The POSIX.1 standard defines extended or additional behavior of certain ANSI C library functions beyond those set forth in ANSI C. The enhanced behavior does not interfere with strict ANSI C compliance.

The following ANSI C functions provide extensions or modified behavior beyond those set forth in ANSI C when used in the POSIX/iX library environment:

- `setlocale()`
- `rename()`
- `abort()`

- ANSI C time functions:
    - `ctime()`
    - `gmtime()`
    - `localtime()`
    - `mktime()`
    - `strftime()`
- `fseek()`
- `exit()`
- `fileno()`
- `fdopen()`
- `fopen()`
- `fclose()`
- `freopen()`
- `fflush()`
- ANSI C functions that read input:
    - `fgetc()`
    - `fgets()`
    - `fread()`
    - `getc()`
    - `getchar()`
    - `gets()`
    - `scanf()`
    - `fscanf()`
- ANSI C functions that write output
    - `fputc()`
    - `fputs()`
    - `fwrite()`
    - `putc()`
    - `putchar()`
    - `puts()`
    - `printf()`
    - `fprintf()`
    - `vprintf()`
    - `vfprintf()`

- `rewind()`
- `perror()`
- `tmpfile()`
- `ftell()`
- `remove()`

Extensions required by POSIX.1 are documented in the library function descriptions found in the *HP C/iX Library Reference Manual* (30026-90001).

Developing Applications Using the MPE/iX Shell and Utilities

Application development using libraries provided with the MPE/iX Developer's Kit must be accomplished through the MPE/iX Shell and Utilities, a command interpreter that provides a set of commands and utilities useful for application development. The MPE/iX Shell is based on the Korn Shell, a command interpreter available on many computer systems.

To invoke the MPE/iX Shell from the MPE/iX Command Interpreter (CI), enter either of the following at the CI prompt:

```
:RUN SH.HPBIN.SYS;INFO="-L"

:SH.HPBIN.SYS -L

:xeq sh.hpbin.sys -l
```

Note The `L` must be entered in uppercase.

For more information about the MPE/iX Shell and Utilities, refer to the following manuals:

- *MPE/iX Shell and Utilities Reference Manual, Volumes 1 and 2* (36431-60001)
- *MPE/iX Shell and Utilities User's Guide* (36431-90002)

Compiling and linking an application that requires libraries available through the MPE/iX Developer's Kit must be accomplished through the `c89` command available in the MPE/iX Shell. For detailed information about using the `c89` command, refer to the *MPE/iX Shell and Utilities Reference Manual, Volumes 1 and 2* (36431-60001).

3

MPE/iX Library Implementation Considerations

This chapter describes important implementation details that you should understand when using functions provided by the MPE/iX library. Implementation details are divided into the following subjects:

- naming file system objects
- files
- directories
- signals
- process management
- access control
- Pipes and FIFOs
- Privileged files in HFS
- Program files in HFS
- HFS aware loader
- Record level locking
- CI Environment Variables
- Symbolic Links
- Device files
- File Emulation
- Read/Write of TAR tapes from the shell

Naming File System Objects

The syntax that the operating system uses to resolve an object name that you specify (either a file or directory) to an actual system object depends upon the interface you are using to access or name the object. A name syntax is a set of rules that defines the structure of valid names for that syntax.

The hierarchical file system (HFS) name syntax used by MPE/iX conforms to object name syntax rules defined by the POSIX.1 standard. (A second name syntax, MPE name syntax, is supported through the MPE/iX Command Interpreter and through system intrinsics.) The POSIX/iX library and the MPE/iX Shell and Utilities interpret object names using only the HFS name syntax when resolving an object name to a system object. You can successfully name any file or directory on your system using HFS name syntax.

The following rules apply when naming files and hierarchical directories using MPE/iX HFS name syntax:

- File and hierarchical directory names can contain alphanumeric characters (A-Z, a-z, 0-9) as well as the dot (.), underscore (_), and dash (-) characters.
- File and hierarchical directory names cannot begin with a dash (-) character.
- File and hierarchical directory names can be up to 255 characters in length; however, certain restrictions apply to file and hierarchical directory names when they are located directly beneath either the root directory or MPE/iX groups. For more information about name restrictions, refer to the sections on files and directories.

For more information about HFS syntax and MPE syntax, refer to *New Features of MPE/iX: Using the Hierarchical File System* (32650-90351).

Files

This section provides an overview of implementation considerations that you should understand when creating, accessing, and managing files through the POSIX/iX library. For more information about how files are created and managed in a POSIX.1 environment, refer to chapter 3, "Files and Directories", in *The POSIX.1 Standard - A Programmer's Guide* (36430-90003).

Additional information about MPE/iX byte-stream files is located in *New Features of MPE/iX: Using the Hierarchical File System* (32650-90351).

POSIX File Types

The following table lists the five file types defined in the POSIX.1 standard as well as their equivalent implementations on MPE/iX:

Table 3-1. MPE/iX Implementations of POSIX File Types

POSIX File Type	MPE/iX File Type
Regular file	Byte-stream file, used to refer to an MPE/iX standard ASCII disk file with a record format of byte-stream.
Directory special file	Hierarchical directory, used to refer to an MPE/iX standard disk file with a file type of directory.
FIFO special file	Supported on 5.0
Block device special file	Not currently implemented on MPE/iX
Character device special file	Not currently implemented on MPE/iX

MPE/iX supports a file whose record format is byte-stream to comply with the regular file behavior defined in the POSIX.1 standard. All files created or opened through POSIX/iX library functions are MPE/iX byte-stream files.

Refer to the section "Directories" for implementation details of MPE/iX HFS directories.

Creating and Opening Files

While MPE/iX supports many file types and file record formats, only MPE/iX byte-stream files can be created or opened using POSIX/iX library functions. Attempts to open an existing file that is not a byte-stream file result in an error, with `errno` set to `EIMPL`. This applies to emulators.

The group ID (GID) of a newly created file or directory is set to the GID of the directory (the parent directory) in which the file is created.

Creating Pipes, FIFOs, and Special Files

Pipes and FIFOs are supported on 5.0 and later systems. Device special files, and read-only file systems are not currently implemented through POSIX/iX interfaces. The standard files are inherited from the parent process, which has them opened as `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO` (defined in the header `<unistd.h>`.

Directory Restrictions

A file created directly under the root directory or directly under an MPE/iX group cannot have a file name that exceeds 16 characters in length. Attempts to create a file whose name exceeds 16 characters in length directly under either the root directory or an MPE/iX group result in an error, with `errno` set to `EIMPL`.

Input/Output Considerations

The following sections describe implementation details associated with POSIX/iX data transfer functions. For more information about POSIX.1 input and output, refer to chapter 4, "Input and Output", in *The POSIX.1 Standard - A Programmer's Guide* (36430-90003).

The POSIX.1 standard does not support the MPE/iX concept of a file limit. MPE/iX provides two facilities for limiting the amount of disk space that a user can have for files:

- file limit
- MPE/iX accounting limits on disk space

File Limits

MPE/iX supports file limits on all objects created on the system to allow users to control the maximum size a that file can attain. Files created through POSIX/iX library functions have file limits. The default file limit for a byte-stream file is two gigabytes when created through POSIX/iX library functions. A file's file limit cannot be manipulated through POSIX/iX functions.

Attempts to write data to a file that would result in that file's size exceeding the file limit result in an error, with `errno` set to `EFBIG`.

The file limit of two gigabytes should rarely, if ever, be reached; however, a user can use MPE/iX CI commands or system intrinsics to set a file limit to a much lower value. If you open a byte-stream file whose file limit has been set to a lower value, the chance of a write error is increased.

MPE/iX Accounting Limits on Disk Space

MPE/iX allows a system administrator to limit the amount of disk space that a user may allocate. MPE/iX disk space limitations can be placed only on MPE/iX accounts and MPE/iX groups; however, a limit placed on an MPE/iX account or MPE/iX group is also imposed on all hierarchical directories and files created at all levels beneath that account or group. No such accounting limits exist for hierarchical directories and files that are not under MPE/iX accounts and MPE/iX groups.

If a process attempts to write data to a file that would result in the disk allocation exceeding MPE/iX disk space limitations, an error is returned, with `errno` set to `EIMPL`.

Additional Implementation Considerations

All POSIX/iX library functions that allow you to specify a pathname return an error and set `errno` to `EIMPL` if you attempt to specify a pathname beginning with two slash characters (//).

The `S_ISUID` and `S_ISGID` bits are not currently implemented.

Access permissions are normally passed or returned through POSIX/iX library functions through a variable of type `mode_t`. Bits of such a variable that are not associated with access control bits must be set to zeros or the function returns an error, with `errno` set to `EIMPL`.

On MPE/iX, the file structure associated with directory streams is implemented using a file descriptor. One effect of this implementation is encountered when using an `exec()` function to execute a file. Because streams are implemented using file descriptors, the file descriptors associated with the parent's streams remain open for the new process image and are counted towards the new process image's `{OPEN_MAX}` count of open file descriptors; however, these file descriptors are inaccessible to the new process image.

MPE/iX supports symbolic links on 5.0 or later systems. MPE/iX supports multiple links to files or hierarchical directories.

All POSIX/iX library functions that allow you to pass a pointer in a parameter return an error and set `errno` to `EFAULT` if the system detects a NULL or bad address in attempting to evaluate the pointer. The only exception to this rule is when a NULL value is a valid value to pass in place of a pointer reference.

Directories

This section provides an overview of implementation considerations that you must understand when creating and managing MPE/iX directories through the POSIX/iX library. For more information about how directories are created and managed in a POSIX.1 environment, refer to chapter 3, "Files and Directories", in *The POSIX.1 Standard - A Programmer's Guide* (36430-90003).

Additional information about MPE/iX directories is located in *New Features of MPE/iX: Using the Hierarchical File System* (32650-90351).

MPE/iX Directories

Beginning with MPE/iX Release 4.5, the directory structure of MPE/iX has been enhanced with the introduction of the MPE/iX hierarchical file system (HFS) directory structure. This has been accomplished by integrating the POSIX.1 hierarchical directory structure within the "classic" MPE directory structure, providing the benefits of both directory structures to existing and new users.

MPE/iX HFS directory services present an integrated view of the file system. Files and hierarchical directories can be created anywhere in the file system.

MPE/iX accounts and MPE/iX groups are special directories that serve as directories while continuing to serve as "classic" MPE accounts and groups.

POSIX/iX directory functions can access any file in the file system hierarchy, including files in the "classic" MPE directory structure, using HFS syntax. For example, both of the following fully qualified file name specifications refer to the same file.

```
/MYACCT/MYGROUP/MYFILE

MYFILE.MYGROUP.MYACCT
```

MPE/iX Directory Features

The following sections describe special features of the MPE/iX root directory, MPE/iX accounts and MPE/iX groups, hierarchical directories, and the dot (`.`) and dot dot (`..`) directories. You need to understand these special features in order to create and manage directories through POSIX/iX library functions.

Root Directory Features

- The MPE/iX root directory (`.`) cannot be renamed, copied, or purged.
- Only users with SM capability can create objects directly under the root directory.
- Access permissions for the root directory are read and execute access for all users and write access for none. Attempts to use the `chmod()` function to remove or change access permissions of the root directory result in an error, with `errno` set to `EINVAL`.
- Names of files and hierarchical directories created directly under the root directory are restricted to 16 characters in length.
- The root directory is restricted to the MPE/iX system volume set.
- The root directory does not contain explicit dot (`.`) and dot dot (`..`) directories; however, dot (`.`) and dot dot (`..`) directory behavior is supported. The dot (`.`) and dot dot (`..`) directories can be opened just like

any other hierarchical directory. File information functions can be used to return information about these directories.

MPE/iX Account Features

- MPE/iX accounts cannot be created, renamed, copied, or purged through POSIX/iX library functions. MPE/iX accounts can be created directly under the root directory only by a user with SM capability using the MPE/iX CI command `NEWACCT`.
- Access permissions for an MPE/iX account are read and execute access for all users and write access for none. Attempts to use the `chmod()` function to remove or change access permissions of an MPE/iX account result in an error, with `errno` set to `EINVAL`.
- When an MPE/iX account name is a component in a pathname, it must be specified in uppercase.
- MPE/iX accounts are restricted to the MPE/iX system volume set.
- An MPE/iX account does not contain explicit dot (.) and dot dot (..) directories; however, dot (.) and dot dot (..) directory behavior is supported. The dot (.) and dot dot (..) directories can be opened just like any other hierarchical directory. File information functions can be used to return information about these directories.
- The user ID (UID) and group ID (GID) associated with an MPE/iX account cannot be modified through POSIX/iX library functions.

MPE/iX Group Features

- MPE/iX groups cannot be created, renamed, copied, or purged through POSIX/iX library functions. MPE/iX groups can be created directly under MPE/iX accounts only by a user with SM capability or a user with AM capability who is a member of that account (whose GID matches the GID of the account). MPE/iX groups are created by MPE/iX CI command `NEWGROUP` and modified by the MPE/iX CI command `ALTGROUP`.
- Default access permissions for MPE/iX groups are read and execute access for all users and write access for none. Attempts to use the `chmod()` function to remove or change access permissions of an MPE/iX group result in an error, with `errno` set to `EINVAL`.

- When an MPE/iX group name is a component in a pathname, it must be specified in uppercase.
- Files and hierarchical directories can be created at any level beneath MPE/iX groups.
- Names of files and hierarchical directories created directly under MPE/iX groups are restricted to 16 characters in length.
- MPE/iX groups (and, indirectly, all files and hierarchical directories at all levels under them) can optionally be assigned to a user volume set.
- MPE/iX accounting limits for disk space apply to both hierarchical directories and files located at all levels under MPE/iX groups.
- An MPE/iX group does not contain explicit dot (.) and dot dot (..) directories; however, dot (.) and dot dot (..) directory behavior is supported. The dot (.) and dot dot (..) directories can be opened just like any other hierarchical directory. File information functions can be used to return information about these directories.
- The user ID (UID) and group ID (GID) associated with an MPE/iX group cannot be modified through POSIX/iX library functions.
- An MPE/iX group must have MPE/iX save access assigned to it before files and hierarchical directories can be created at any level under it.

Hierarchical Directory Features

- Hierarchical directories cannot be renamed through POSIX/iX library functions.
- Users can define and modify access permissions for hierarchical directories through POSIX/iX library functions.
- The group ID (GID) of a hierarchical directory is inherited from its parent directory. The user ID (UID) of a hierarchical directory is inherited from the effective UID of the process that created it. The UID and GID of a hierarchical directory can be modified using the `chown()` function.
- Hierarchical directories (and all objects under them) that are not under MPE/iX accounts and MPE/iX groups are restricted to the MPE/iX system volume set.

- Names of hierarchical directories and files located directly under either the root directory or MPE/iX groups are restricted to 16 characters in length.
- Names of hierarchical directories that are not directly under either the root directory or MPE/iX groups are restricted to 255 characters in length. This limit is defined by `{NAME_MAX}`, found in the header `<limits.h>`.
- Hierarchical directories contain explicit dot (.) and dot dot (..) directory entries.

Dot and Dot Dot Directory Features

When a hierarchical directory is first created by the `mkdir()` function, two special directory entries are placed in the hierarchical directory:

- The dot (.) directory entry is an alternative way to specify a current directory without having to use a formal directory name.
- The dot dot (..) directory is an alternative way to specify a current directory's parent directory without having to use a formal directory name.

The dot (.) and dot dot (..) directories provide additional navigation aids to a process. Applications using the dot (.) and dot dot (..) directories to express current directory or parent directory need not be concerned with their absolute location on the system. The use of these directory names in pathname resolution increases the portability of applications to any location in a file system.

The dot (.) and dot dot (..) directories can be opened just like any other hierarchical directory. File information functions can be used to return information about these directories.

These two directories cannot be explicitly purged from their parent directory except by purging the parent directory. For example, an attempt to use `rmdir()` to purge the dot (.) or dot dot (..) directory results in an error, with `errno` set to `EIMPL`.

Note The dot (.) and dot dot (..) directories are not found explicitly under the root directory or under MPE/iX accounts and MPE/iX groups; however, dot (.) and dot dot (..) behavior is supported.

Additional Implementation Considerations

The return type of the `rewinddir()` function is implemented as `int` in order to return a value of -1 indicating an error. The POSIX.1 standard calls for no value to be returned `(void)`. A strictly conforming POSIX.1 application should not evaluate values returned by `rewinddir()`.

If an entry is purged from or added to a directory after the most recent call to `opendir()` or `rewinddir()`, subsequent returns from `readdir()` accurately reflect the current state of the directory.

The `unlink()` function cannot be used to purge hierarchical directories. Instead, use `rmdir()` to purge hierarchical directories.

Access Control

This section provides an overview of implementation considerations that you should understand when using access control features as they are implemented in the POSIX/iX library.

MPE/iX security features have been enhanced to provide full support for security features defined by the POSIX.1 standard. MPE/iX supports additional access security features, but they remain largely transparent to your application.

You may need to understand these underlying access security features if your application accesses files that were not created by the POSIX/iX library functions, or if the security of those files was modified by MPE/iX CI commands or system intrinsics.

For more information about how file access permissions are used in a POSIX.1 environment, refer to chapter 3, "Files and Directories", in *The POSIX.1 Standard - A Programmer's Guide* (36430-90003).

For more information about MPE/iX implementation of POSIX.1 security standards, refer to *User's Guide to MPE/iX Security* (32650-90472).

MPE/iX Access Control Definitions

Access permissions defined by the POSIX.1 standard are fully supported in the POSIX/iX library through the use of the MPE/iX access control definition (ACD) facility. POSIX.1 security is fully integrated with MPE/iX security. Except in cases described below, ACD access control remains transparent when accessed through POSIX/iX library functions. The POSIX/iX library automatically provides translation between the POSIX.1 view of access permission and the MPE/iX view of access permission.

ACDs are required for the following file system objects:

- All hierarchical directories.
- All files under hierarchical directories.
- All files directly under MPE accounts.
- All files directly under MPE/iX groups where the file GID does not match the GID of the account and group in which the file resides.

A file or hierarchical directory created by POSIX/iX library functions is automatically assigned an ACD.

For more information about MPE/iX ACDs, refer to *New Features of MPE/iX: Using the Hierarchical File System* (32650-90351).

Mapping Between POSIX.1 and ACD Access Permissions

The following table describes the correspondence between POSIX.1 file access permissions and MPE/iX ACD access permissions.

Table 3-2.
Mapping between POSIX.1 and ACD File Access Permissions

POSIX.1 Access Permissions	ACD Access Permissions
Read	ACD read (R) access
Write	ACD write (W) access
Execute	ACD execute (X) access

The following table describes the correspondence between POSIX.1 directory access permissions and MPE/iX ACD access permissions.

Table 3-3.
Mapping between POSIX.1 and ACD Directory Access Permissions

POSIX.1 Access Permissions	ACD Access Permissions
Read	ACD read directory entries (RD) access
Write	Both ACD create directory entries (CD) and ACD delete directory entries (DD) access
Execute	ACD traverse directory entries (TD) access

Write access to a directory is implemented by two MPE/iX ACD access modes, create directory entry (CD) and delete directory entry (DD). Setting or modifying write access permission to a directory using POSIX/iX library functions always modifies both ACD access modes. Both CD and DD access modes must be specified in order for a POSIX/iX library function to have write access to a hierarchical directory.

MPE/iX CI commands and system intrinsics allow you to remove either the CD or DD access mode. When this occurs, both `fstat()` and `stat()` indicate that write access permission is no longer granted to the process; however, if only DD access is specified, a process can delete directory entries but cannot create directory entries. Likewise, if only CD access is specified, a process can create directory entries but cannot delete directory entries.

Mapping Between POSIX.1 and ACD File User Classes

The following table describes the correspondence between the POSIX.1 file user classes and MPE/iX ACD user specifications.

Table 3-4. Mapping between POSIX.1 and ACD User Classes

POSIX.1 File Classes	ACD User Specifications
File owner	`$OWNER`
File group	`$GROUP` and `$GROUP_MASK`
File other	`@.@`

The `$OWNER` ACD entry specifies the owner of the file or directory. A user is a file or directory owner if the user's effective UID matches the UID of the file.

The `$GROUP` ACD entry specifies the group members of the file or directory. A user is a file or directory group member if the user's effective GID matches the GID of the file.

The `$GROUP_MASK` entry restricts all ACD entries except for `$OWNER` and `@.@`. In this case, if a user matches a *user.account* entry, an *@.account* entry, or a `$GROUP` entry, the user is granted only the access permissions that appear in both the matching entry and the `$GROUP_MASK` entry. An ACD with a `$GROUP_MASK` entry must also contain a `$GROUP` entry. The `$GROUP_MASK` entry serves to integrate the POSIX definition of security with the more robust security provided by MPE/iX ACDs.

The `@.@` ACD entry specifies the file other class members of the file or directory who are not members of the file owner class or the file group class.

A file or hierarchical directory created by POSIX/iX library functions is automatically assigned an ACD containing the four MPE/iX user specifications `$OWNER`, `$GROUP`, `$GROUP_MASK`, and `@.@`. The access permissions associated with each of the four user specifications are initialized from the *mode* parameter and modified by the calling process's file creation mask. The access modes associated with the `$GROUP_MASK` entry are initialized to the same access modes associated with the `$GROUP` entry.

The following example shows the correspondence between a POSIX view of access permissions and the underlying ACD.

Example of POSIX file security and underlying ACD

File Owner Class	File Group Class	File Other Class
rwx	r-x	--x
R,W,X,RACD:$OWNER	R,X,RACD:$GROUP R,X, RACD:$GROUP_MASK	X:@.@

If the ACD Has Been Modified or Removed

If you are accessing files that were not created through POSIX/iX library functions, or whose security was modified by MPE/iX commands or system intrinsics, the ACDs may be missing any or all of the four ACD user specification entries `$OWNER`, `$GROUP`, `$GROUP_MASK`, and `@.@`. ACDs missing any of these four ACD user specification entries still support access control as defined by the POSIX.1 standard since they can be regarded as containing default values for the missing ACD entries.

Summary of fstat() and stat() Behavior

MPE/iX uses the rules described in the following sections when determining the access permissions returned by calls to the `fstat()` and `stat()` functions.

If the File or Directory has an ACD

If the ACD contains a `$OWNER` entry, `fstat()` and `stat()` return for the file owner class the access permissions associated with the `$OWNER` entry. If the ACD contains no `$OWNER` entry, `fstat()` and `stat()` return for the file owner class the access permissions of both read and write access. Execute access is also returned for the file owner class if any ACD entry specifies execute access to the file.

If the ACD contains an @.@ entry, `fstat()` and `stat()` return for the file other class the access permissions associated with the @.@ entry. If the ACD contains no @.@ entry, `fstat()` and `stat()` return for the file other class access permissions of NONE.

If the ACD contains a `$GROUP_MASK` entry, `fstat()` and `stat()` return for the file group class the access permissions associated with the `$GROUP_MASK` entry.

If the ACD contains no `$GROUP_MASK` entry, `fstat()` and `stat()` return for the file group class the access permissions resulting from ORing all file group class members (all *user.account* entries, *@.account* entries, and the `$GROUP` entry). If the `$GROUP` entry and the *@.account* entry that corresponds to the file GID are both missing, and an @.@ entry exists, the permissions associated with the @.@ entry are included in the calculation.

If the File Does Not Have an ACD

If the file does not have an ACD, `fstat()` and `stat()` return access permissions resulting from an evaluation of the MPE/iX file system security matrix. Evaluation occurs in the following manner:

- The file owner class access permissions returned by `fstat()` and `stat()` indicate both read and write access. Execute access is also indicated if any user has execute access to the file.
- The file group class access permissions returned by `fstat()` and `stat()` are the result of ORing together the access permissions associated with the file system security matrix AC and ANY user specifications.
- The file other class access permissions returned by `fstat()` and `stat()` are the access permissions associated with the file system security matrix ANY user specification.

The Root Directory, MPE/iX Accounts, and MPE/iX Groups

For the root directory and MPE/iX accounts, the file owner class, file group class, and file other class access permissions returned by `fstat()` and `stat()` indicate read and execute access.

For MPE/iX groups, the file owner class and file group class access permissions returned by `fstat()` and `stat()` indicate read, write, and execute access. The file other access permissions indicate read and execute access.

Summary of chmod() Behavior

MPE/iX uses the rules described in the following sections when determining how the `chmod()` function modifies access permissions.

If the File or Directory has an ACD

If the ACD contains the `$GROUP` entry, and does not contain *user.account* or *@.account* entries, the `$GROUP` entry is assigned the file group class access permissions passed by `chmod()`, plus RACD access.

If the ACD contains the `$GROUP` entry as well as *user.account* or *@.account* entries, the `$GROUP` entry is not affect by `chmod()`. In this case, only the `$GROUP_MASK` is assigned the file group class access permissions passed by `chmod()`, plus RACD access.

If the ACD does not contain the `$GROUP` entry, and does not contain *user.account* or *@.account* entries, the `$GROUP` entry is created with the file group class access permissions passed by `chmod()`, plus RACD access.

If the ACD does not contain the `$GROUP` entry but contains user.account entries and/or @.account entries, the following rules apply, in order of precedence:

- If an *@.account* entry exists where *account* matches the GID of the file, the `$GROUP` entry is created with the access permissions of that *@.account* entry.
- If no *@.account* entry exists where *account* matches the GID of the file, and an @.@ entry exists, `$GROUP` is created with the access permissions of the @.@ entry.
- If no *@.account* entry exists where *account* matches the GID of the file, and no @.@ entry exists, `$GROUP` is created with access permissions of NONE.

If the ACD contains the `$GROUP_MASK` entry, the `$GROUP_MASK` entry is assigned the file group class access permissions passed by `chmod()`, plus RACD access. If the ACD does not contain the `$GROUP_MASK` entry, the `$GROUP_MASK` entry is created with the file group class access permissions passed by `chmod()`, plus RACD access.

If the ACD contains a `$OWNER` entry, the `$OWNER` entry is assigned the file owner class access permissions passed in `chmod()`, plus RACD access. If the ACD contains no `$OWNER` entry, a `$OWNER` entry is created with the file owner class access permissions passed in `chmod()`, plus RACD access.

If the ACD contains the @.@ entry, the @.@ entry is assigned the file other class access permissions passed by `chmod()`, plus RACD access. If the ACD contains no @.@ entry, the @.@ entry is created with the file other class access permissions passed in `chmod()`, plus RACD access.

If the File Does Not Have an ACD

When `chmod()` is invoked on a file that does not have an ACD, an ACD is created with the following user specifications and access permissions:

- The `$OWNER` entry is assigned the file owner class access permissions passed by `chmod()`, plus RACD access.
- Both the `$GROUP` and `$GROUP_MASK` entries are assigned the file group class access permissions passed by `chmod()`, plus RACD access.
- The @.@ entry is assigned the file other class access permissions passed by `chmod()`, plus RACD access.

The Root Directory, MPE/iX Accounts, and MPE/iX Groups

Attempts to use `chmod()` to modify access permissions of the root directory, MPE/iX accounts, or MPE/iX groups result in an error, with `errno` set to `EINVAL`.

Determining a Process's Access to a File or Directory

For information about MPE/iX implementation of POSIX.1 security standards, refer to *User's Guide to MPE/iX Security* (32650-90472).

Refer to chapter 9, "Handling Security on MPE/iX", in *New Features of MPE/iX: Using the Hierarchical File System* (32650-90351) for a thorough description of how MPE/iX determines a process's access to a file or directory.

Returning Information about Access Permissions

An additional MPE/iX ACD access permission, read ACD (RACD) access, is used to restrict a user from reading access permissions of a directory or file that is assigned an ACD. The POSIX/iX library does not allow manipulation of the RACD access permission. By default, all users are given RACD access to all objects created through POSIX/iX library functions. This default allows queries of directory and file access permissions to occur through the `stat()` and `fstat()` functions without error.

MPE/iX provides facilities outside the POSIX/iX library to manipulate the RACD access permission of a directory or file. If a process attempts to invoke the `fstat()` or `stat()` function on a directory or file with an ACD that does not allow RACD access to that process, both functions return an error, with `errno` set to `EPERM`.

You can modify RACD access to a file or directory using the MPE/iX CI command `ALTSEC`.

MPE/iX Save Files (SF) Capability

The user associated with a process must have save files (SF) capability to create an entry in a file. In POSIX.1 terminology, SF capability acts as an additional access control mechanism. A process must, therefore, have SF capability to successfully create files or hierarchical directories. The SF capability is an MPE/iX capability assigned to a user through the MPE/iX CI commands `NEWUSER` or `ALTUSER`.

MPE/iX Lockwords

MPE/iX provides an additional file security feature, file lockwords, that is not accessible through POSIX/iX library functions; however, file lockword security is suppressed for all files and directories that contain ACDs. Lockwords can be encountered only on files located directly under MPE/iX groups.

Attempts to open an existing file that has an MPE/iX lockword result in an error, with `errno` set to `EACCESS`. Attempts to modify the access permissions of an existing file that has an MPE/iX lockword also result in an error.

For more information about MPE/iX lockwords, refer to *User's Guide to MPE/iX Security* (32650-90472).

Signals

This section provides an overview of implementation considerations that you must understand when using signals as they are implemented in the POSIX/iX library. For more information about how signals are used in a POSIX.1 environment, refer to chapter 5, "Signals", in *The POSIX.1 Standard - A Programmer's Guide* (36430-90003).

Supported Signal Functions

All signal functions defined by the POSIX.1 standard are implemented in the POSIX/iX library. While the ANSI C functions `signal()` and `raise()` are provided in the POSIX/iX library, they are not part of the POSIX.1 standard. A strictly conforming POSIX application should not use them.

Signal Descriptions

Table 3-5 describes the signal constants declared in the `<signal.h>` header that are used by a process to refer to the signals that occur on the system. Also noted are the default action taken by the system when the signal is delivered, whether the signal is required for POSIX.1 conformance, and any MPE/iX implementation details.

Table 3-5. POSIX/iX Signals

Constant	Default Action	Description and Implementation Details
SIGABRT	Abnormal termination	Abnormal termination signal (initiated by the `abort()` function). Required for POSIX.1 conformance.
SIGALRM	Abnormal termination	Timeout signal (initiated by the `alarm()` function). Required for POSIX.1 conformance.
SIGBUS	Abnormal termination	Address violation. `SIGBUS` is not required for POSIX.1 conformance, but is a signal commonly used on UNIX™-based systems. A POSIX.1-conforming application should not rely upon the generation of this signal by the system. The results are undefined if the system generates this signal while the signal is either blocked, ignored, or has a signal-handling function that returns normally.
SIGFPE	Abnormal termination	Erroneous arithmetic operation, such as division by zero or a floating-point exception. The results are undefined if the system generates this signal while the signal is either blocked, ignored, or has a signal-handling function that returns normally. Required for POSIX.1 conformance.
SIGHUP	Abnormal termination	Hang-up detected on a controlling terminal or death of a controlling process. Required for POSIX.1 conformance.
SIGILL	Abnormal termination	Detection of an invalid or illegal hardware instruction (not reset when caught). Required for POSIX.1 conformance.
SIGINT	Abnormal termination	Interactive interrupt signal. Required for POSIX.1 conformance.

Table 3-5. POSIX/iX Signals (continued)

Constant	Default Action	Description and Implementation Details
SIGKILL	Abnormal termination	Termination signal (cannot be caught or ignored). If an application attempts to change the default action associated with **SIGKILL**, the attempt is ignored without error. Required for POSIX.1 conformance.
SIGPIPE	Abnormal termination	Write on a pipe with no readers. Required for POSIX.1 conformance.
SIGPOLL	Ignore	Streams poll signal. Also known as **SIGIO**.
SIGQUIT	Abnormal termination	Interactive termination signal. Required for POSIX.1 conformance.
SIGSEGV	Abnormal termination	Detection of an invalid or illegal memory reference. The results are undefined if the system generates this signal while the signal is either blocked, ignored, or has a signal-handling function that returns normally. Required for POSIX.1 conformance.
SIGTERM	Abnormal termination	Software termination signal (initiated by the **kill()** function). Required for POSIX.1 conformance.
SIGURG	Ignore	Urgent condition in I/O channel.
SIGUSR1	Abnormal termination	Reserved as application-defined signal #1. Required for POSIX.1 conformance.
SIGUSR2	Abnormal termination	Reserved as application-defined signal #2. Required for POSIX.1 conformance.

Table 3-5. POSIX/iX Signals (continued)

Constant	Default Action	Description and Implementation Details
Job Control Signals		
`SIGCHLD`	Ignore the signal	Child process stopped or terminated. Required for POSIX.1 conformance.
`SIGCONT`	Continue if stopped; otherwise, ignore	Continue if stopped. This signal is never generated by the system. Required for POSIX.1 conformance.
`SIGSTOP`	Stop the process	Stop signal (cannot be caught or ignored). If an application attempts to change the default action associated with `SIGSTOP`, the attempt is ignored without error. This signal is never generated by the system. Required for POSIX.1 conformance.
`SIGTSTP`	Stop the process	Interactive stop signal. Because job control is not currently implemented, your application should not rely upon the generation of this signal by the system. Required for POSIX.1 conformance.
`SIGTTIN`	Stop the process	Read from the controlling terminal attempted by a member of a background process group. Because job control is not currently implemented, your application should not rely upon the generation of this signal by the system. Required for POSIX.1 conformance.
`SIGTTOU`	Stop the process	Write to the controlling terminal attempted by a member of a background process group. Because job control is not currently implemented, your application should not rely upon the generation of this signal by the system. Required for POSIX.1 conformance.

Additional Implementation Considerations

On MPE/iX, signals cannot be delivered to a process while that process is executing system code. The signal remains pending until control returns to the calling process. A sending process cannot rely on timely delivery of a signal if the target process is executing system code. For example, if a signal is generated for a process when the process is executing a call to `read()` or `write()`, the signal remains pending until the function call returns either after successful transfer of data or when an error is encountered.

The ANSI C `signal()` function is implemented in the POSIX/iX library as a call to `sigaction()`; however, the `signal()` function is considered by the POSIX.1 standard to be incompatible with the POSIX.1 `sigaction()` function. A strictly conforming POSIX.1 application must not use the `signal()` function.

The `sigaction()` function can return and reinstall a signal action that was originally installed by `signal()`; however, the structure that `sigaction()` returns in *oact* cannot be reliably examined by the calling process. If this same signal action is later reinstalled, without modification, by another call to `sigaction()`, the result is as if the original call to `signal()` were repeated.

If multiple occurrences of a signal are generated while that signal is blocked and pending, each occurrence of the signal is left pending. If the signal is later unblocked, multiple instances of that signal can be delivered to the process.

A signal that is both blocked and ignored for a calling process remains pending if generated. Calls to the `sigpending()` function return signals that are both blocked and ignored. When the signal is no longer blocked, it is discarded when delivery is attempted.

Setting `SIGCHLD` to `SIG_IGN` has no effect on the operation of the `wait()` and `waitpid()` functions. A strictly conforming POSIX.1 application must not set the action associated with `SIGCHLD` to `SIG_IGN`.

When using the `kill()` function, the value -1 is not a valid value for passing to the *pid* parameter. If a -1 is passed in the *pid* parameter, `kill()` returns an error and sets `errno` to `EINVAL`.

On MPE/iX, a process's real user ID, effective user ID, and saved set-user-ID are always identical. In addition, `{SAVED_SET_IDS}` is always defined.

If the *sigmask* parameter of the `sigsuspend()` function is set to NULL, the process is suspended with the current signal mask. This implementation is considered an extension to the POSIX.1 standard. A strictly conforming POSIX application should pass in the *sigmask* parameter of the `sigsuspend()` function the current signal mask returned by a successful call to `sigprocmask()` where *set* is set to NULL.

POSIX/iX library functions that have parameters that pass or return pointers can return an error and set `errno` to `EFAULT` if a NULL or a bad address is passed.

Process Management

This section provides an overview of implementation considerations that you must understand when creating and managing processes as they are implemented in the POSIX/iX library. For more information about how processes are created and managed in a POSIX.1 environment, refer to chapter 6, "Process Creation and Synchronization", in *The POSIX.1 Standard - A Programmer's Guide* (36430-90003).

The implementation and behavior of processes created through the POSIX/iX library conform in most respects to the POSIX.1 standard. In most cases, underlying MPE/iX process features are transparent to a POSIX.1 application; however, there are some MPE/iX features outside the scope of the POSIX.1 standard that cannot be hidden from your application. These additional implementation features must be taken into account when you are creating and managing processes through the POSIX/iX library.

Note Users need PH capability when running a program in an HFS directory that spawns childprocesses.

Creating a New Process

The following implementation considerations must be understood when using `fork()` or an `exec()` function to create or execute processes.

- The executable file must have an MPE/iX file code of `NMPRG`.
- The MPE/iX process handling (PH) capability must be appropriately assigned. Process handling capability is described in the section "MPE/iX Process Handling Capability."

An executable file that is compiled and linked using the `c89` command (available through the MPE/iX Shell) always creates an executable file with an MPE/iX file code of `NMPRG`. Attempts to use an `exec()` function to execute a file that has a file code of anything other than `NMPRG` results in an error, with `errno` set to `ENOEXEC`.

To determine whether a file has an MPE/iX file code of `NMPRG`, you must use the MPE/iX CI command `LISTFILE`. For more information about using the `LISTFILE` command, refer to *MPE/iX Commands Reference* manual.

MPE/iX Process Handling Capability

By default, MPE/iX restricts an application's ability to spawn multiple processes. The MPE/iX process handling (PH) capability allows the creation of multiple processes.

There are two levels of restrictions that apply if an application wishes to invoke the `fork()` function or `exec()` functions:

- The executable file must be linked with PH capability. The `c89` command available through the MPE/iX Shell automatically assigns PH capability to files at link time.
- If the executable file resides in an MPE/iX group, that group must have PH capability in order to execute the file. PH capability is assigned by a user with either SM or AM capability to an MPE/iX group using the MPE/iX CI commands `NEWGROUP` (when the group is created) or `ALTGROUP` (when the group exists).
- If an executable file is located in a hierarchical directory, the user attempting to execute the file must have PH capability assigned using the `ALTUSER` command.

Inherited Process Attributes

Because processes created through POSIX/iX library functions reside in an MPE/iX process environment, certain MPE/iX process attributes are inherited by a process created by `fork()`; however, these MPE/iX process attributes are not visible in the POSIX/iX environment.

For example, the following MPE/iX process attributes are inherited by a new process created by `fork()`:

- process priority
- capabilities
- stack size
- heap size

Also, some MPE/iX process attributes that are not defined by the POSIX.1 standard are not inherited by the child process. The lack of these attributes does not affect the behavior of a process created in the POSIX/iX environment.

Process Termination

On MPE/iX, if a parent process terminates without waiting for all of its child processes to terminate, the resulting "orphaned" child processes are terminated immediately prior to termination of the parent process. The implementation does not allow orphaned child processes to be adopted by a system process. Your application should not rely upon orphaned child processes being adopted by a system process.

Additional Implementation Considerations

No user process can be a controlling process. Only system processes, such as the MPE/iX Command Interpreter (CI), are allowed to be controlling processes.

The controlling terminal is not disassociated from the session when a user process terminates. The controlling terminal is associated with the MPE/iX CI session that invokes the application. The controlling terminal is only disassociated when the MPE/iX session is ended (when the user logs off the system using the `BYE` command).

CPU time accounting information accrued by process is not made available to the parent process through the `wait()` and `waitpid()` functions. A zero is always returned.

Attempts to use the `fork()` or `exec()` function to create a new process fails if the calling process

- has active switches to MPE/iX compatibility mode (CM) code
- has set critical mode
- has outstanding NOWAITIO
- is holding an operating system internal resource (SIR)

4

POSIX/iX Library Function Descriptions

This chapter describes POSIX/iX library functions defined in the POSIX.1 standard. Function descriptions are arranged alphabetically.

access

Check file accessibility

Syntax

```
#include <unistd.h>
int access (const char *path, init amode);
```

Parameters

path The pathname of a file.

amode One of the following file access permissions.

The bitwise inclusive OR of the following access permission constants to be checked:

Access Permissions	**Descriptions**
R_OK	Test for read permission.
W_OK	Test for write permission.
X_OK	Test for execute or search permission.

Or the existence test (F_OK)

Other values of the *amode* argument are ignored.

Description

The access permissions of the filenamed *path* is checked by the `access()` function. The `path` argument for file access permissions is indicated by `amode` based on the real (not effective) user ID (UID) and group ID (GID).

The `amode` value is the bitwise inclusive OR of the access permissions or the existence test checking if the file exists or not.

The three access permissions are checked individually, if they need to be checked at all. If the process has appropriate privileges, execute file permission will be granted.

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to one of the following values:

`E2BIG`	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of `{ARG_MAX}` (defined in `<limits.h>`).
	ACTION	Reduce the size of the argument list or environment list or both.
`EACCES`	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account, or the pathname begins with two slashes.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.

`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.

`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.

See Also

`chmod()`, `stat()`, `<unistd.h>`

alarm

Schedules a `SIGALRM` signal.

Syntax

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

Parameters

seconds	The number of real-time seconds to wait before generating a `SIGALRM` signal. A zero cancels any previously scheduled alarm request.

Return Values

0	Success. There is no previously scheduled alarm.
>0	Success. The number of seconds remaining on a previously scheduled alarm is returned.

Description

The `alarm()` function causes the operating system to generate a `SIGALRM` signal for the calling process after the number of real-time seconds specified by the *seconds* parameter have elapsed. Operating system scheduling delays may prevent delivery of the signal until after the specified time.

Only one `SIGALRM` signal can be scheduled at a time. Any previously scheduled alarm is cancelled by the current alarm. A previously scheduled alarm is cancelled by passing zero in the *seconds* parameter.

Implementation Considerations

Currently, `alarm()` does not cause a read timeout.

Errors

None.

See Also

`fork()`, `pause()`, `sigaction()`, `<signal.h>`, POSIX.1 (Section 3.4.1)

chdir

Changes the current working directory.

Syntax

```
#include <unistd.h>
int chdir (const char *pathname);
```

Parameters

pathname A pointer to a string containing the pathname of the directory to be the current working directory. The pathname must be terminated by a null character.

Return Values

0	Success.
-1	An error occurred. The current working directory is not changed, and `errno` is set to indicate the error condition.

Description

The `chdir()` function causes the directory specified by *pathname* to be the current working directory of the calling process. The current working directory is the directory used by a process in resolving pathnames not beginning with a slash character (/).

If `chdir()` fails, the current working directory remains unchanged and a -1 is returned.

Implementation Considerations

Refer to the `EFAULT`, `EIMPL`, and `ESYSERR` error descriptions below.

The `chdir()` function does not affect the logon MPE/iX group or MPE/iX account against which CPU and connect time are accumulated, nor does `chdir()` alter the set of accessible files.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	The calling process does not have search permission to a component of the pathname.
	ACTION	Make sure that the calling process has search permission to all components of the pathname.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	The pathname begins with two slash characters (//).
	ACTION	Do not begin pathnames with two slash characters (//).
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	The specified directory does not exist or *pathname* points to an empty string.
	ACTION	Specify an existing directory name.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`getcwd()`, POSIX.1 (Section 5.2.1)

chmod

Changes file access permissions.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (const char *pathname, mode_t mode);
```

Parameters

pathname A pointer to a string containing the pathname of a file or directory whose access permissions are to be modified. The pathname must be terminated by a null character.

mode New access permissions. Access permission bits are set by ORing any combination of the following macros:

S_IRWXU	Set file owner class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRUSR	Set file owner class read permission bit.
S_IWUSR	Set file owner class write permission bit.
S_IXUSR	Set file owner class execute (if a file) or search (if a directory) permission bit.
S_IRWXG	Set file group class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRGRP	Set file group class read permission bit.
S_IWGRP	Set file group class write permission bit.
S_IXGRP	Set file group class execute (if a file) or search (if a directory) permission bit.
S_IRWXO	Set file other class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IROTH	Set file other class read permission bit.

`S_IWOTH`	Set file other class write permission bit.
`S_IXOTH`	Set file other class execute (if a file) or search (if a directory) permission bit.

Unused bits of the *mode* parameter not associated with access permissions must contain zeros, or an error occurs.

Return Values

0	Success.
-1	An error occurred. Access permission bits are not changed, and `errno` is set to indicate the error condition.

Description

The `chmod()` function sets the file access permission bits of the file or directory named in *pathname* to the bits specified in *mode*. Access permissions can be modified only when one of the following conditions is true:

- The user associated with the calling process is the file owner (a user whose effective UID matches the UID of the file).
- The user associated with the calling process has appropriate privileges, defined to be one of the following:
  - a user whose GID matches the GID of the file and who has the MPE/iX account manager (AM) user capability
  - a user who has the MPE/iX system manager (SM) user capability

Upon successful completion, `chmod()` marks for update the `st_ctime` time field of the file.

Implementation Considerations

Refer to the `EIMPL`, `EINVAL`, and `EFAULT` error descriptions below.

The `S_ISUID` and `S_ISGID` bits are not currently supported.

Changes to file access permission bits do not affect access to *pathname* through open file descriptors already associated with *pathname* at the time of the `chmod()` call.

If bits in *mode* other than access permission bits are set to a nonzero value, an error is returned and access permission bits are not changed.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	The calling process does not have search permission to a component of the pathname.
	ACTION	Make sure that the calling process has search permission to all component directories in the pathname.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use *pathname*, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	One of the following: ■ The pathname begins with two slash characters (//). ■ The unused bits of the *mode* parameter do not contain zeros.
	ACTION	One of the following: ■ Do not begin pathnames with two slash characters (//). ■ Set to zero all *mode* bits that are not access permission bits.
`EINVAL`	CAUSE	The *pathname* parameter specified the root directory, an MPE/iX account, or an MPE/iX group.
	ACTION	Do not attempt to change the access permission bits of the root directory, an MPE/iX account, or an MPE/iX group.

`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	The specified file does not exist, or *pathname* points to an empty string.
	ACTION	Specify an existing filename.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.

See Also

`chown()`, `mkdir()`, `stat()`, `fstat()`, `<sys/stat.h>`, POSIX.1 (Section 5.6.4)

chown

Changes the owner and group of a file.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
int chown (const char *pathname, uid_t owner,
 gid_t group);
```

Parameters

pathname A pointer to a string containing the pathname of a file whose user ID and group ID are to be modified. The pathname must be terminated by a null character.

owner The new owner (user ID) of the file.

group The new group ID of the file.

Return Values

0 Success.

-1 An error occurred. The file's owner and group ID are not changed, and `errno` is set to indicate the error condition.

Description

The `chown()` function changes the user ID (UID) of the specified file to *owner* and the group ID (GID) of the file to *group*.

In order to change the UID of a file, the user associated with the calling process must be one of the following:

- The file's account manager (a user whose GID matches the GID of the file and who has the MPE/iX account manager (AM) user capability). In this case, *owner* must specify a UID belonging to a user assigned to the account manager's own account, and *group* must specify the account manager's own effective GID.

- A system manager (a user who has the MPE/iX system manager (SM) user capability). In this case, *owner* can specify any UID existing in the user database.

In order to change the GID of a file, the user associated with the calling process must be one of the following:

- The file owner (a user whose effective UID matches the UID of the file). In this case, *owner* must specify the file's UID and *group* must specify the file owner's own GID.
- A user with appropriate privileges, defined to be one of the following:
  - The file's account manager (a user whose GID matches the GID of the file and who has the MPE/iX account manager (AM) user capability). In this case, *owner* must specify the UID of a user assigned to the account manager's own account, and *group* must specify the account manager's own effective GID.
  - A system manager (a user who has the MPE/iX system manager (SM) user capability). In this case, *group* can specify any GID existing in the group database.

Upon successful completion, `chown()` marks for update the `st_ctime` time field of the file.

Implementation Considerations

Refer to the `EIMPL` and `EFAULT` error descriptions below.

The `S_ISUID` and `S_ISGID` bits are not supported.

The `{_POSIX_CHOWN_RESTRICTED}` constant is always in effect for *pathname*.

You cannot modify the GID of the root directory, MPE/iX accounts, or MPE/iX groups.

An object's owner, its account manager(s), and system managers have different abilities to assign UID and GID values. A system manager (user with MPE/iX SM capability) can specify any positive UID or GID value defined in the user or group databases. An account manager (user with MPE/iX AM capability) can specify the UID of any user belonging to the account manager's account, but can only specify the GID associated with the account manager's own effective GID. File owners lacking SM or AM capability cannot change a file's UID, but can change a file or directory's GID to their own effective GID.

Changing an object's file owner ID (UID) or file group ID (GID) changes access control for that file or directory. File owners of files and directories can also change the access permissions granted to the object. Changing an object's UID or GID also changes the file owner or file group referenced by `$OWNER` and `$GROUP` entries in the ACD associated with the file or directory.

An ACD is automatically assigned to a file if the file lacks an ACD and the *group* parameter specifies a different GID than the GID associated with the MPE/iX account in which the file resides. The new ACD grants all access to the file owner and RACD access to all others.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	The calling process does not have search permission to a component of the pathname.
	ACTION	Make sure that the calling process has search permission to all component directories in the pathname.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	One of the following: ■ An attempt was made to change the UID or GID of the root directory, an MPE/iX account, an MPE/iX group, an output spool file, or a system-defined file. ■ The pathname begins with two slash characters (//).
	ACTION	One of the following: ■ Do not attempt to change the UID or GID of the root directory, an MPE/iX account, an MPE/iX group, an output spool file, or a system-defined file. ■ Do not begin pathname with two slash characters (//).

`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	The specified file does not exist, or *pathname* points to an empty string.
	ACTION	Specify an existing filename.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EINVAL`	CAUSE	The *owner* parameter or *group* parameter specified an invalid or unsupported value.
	ACTION	Specify a valid and supported value.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.

See Also

`chmod()`, POSIX.1 (Section 5.6.5)

close

Closes a file.

Syntax

```
#include <unistd.h>
int close (int fildes);
```

Parameters

fildes An open file descriptor.

Return Values

0 Success. The file is closed.

-1 An error occurred. The file is not closed, and `errno` is set to indicate the error condition.

Description

The `close()` function closes the file specified by *fildes*. Upon the close, all record locks held by the calling process on the file associated with *fildes* are removed.

When all file descriptors associated with an open file description have been closed, the open file description is freed and is no longer accessible.

If the link count of the file is zero upon closing, when all open file descriptors associated with the file are closed, the file is purged from the system.

The `close()` function updates the following file time fields to the current time:

- All time fields that have been previously marked for update. All update marks are removed.
- The `st_atime` time field.
- The `st_mtime` time field only if the file was opened `O_WRONLY` or `O_RDWR`.

Implementation Considerations

Refer to the `ESYSERR` error description below.

Signals generated for the calling process during the execution of `close()` are deferred from delivery until the completion of `close()`.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *fildes* parameter is not a valid open file descriptor.
	ACTION	Check to see if *fildes* has been altered or is not initialized.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`creat()`, `dup()`, `execl()`, `execv()`, `fork()`, `open()`, `unlink()`, POSIX.1 (Section 6.3.1)

closedir

Closes a directory stream.

Syntax

```
#include <sys/types.h>
#include <dirent.h>
int closedir (DIR *dirp);
```

Parameters

dirp A pointer to a structure of type `DIR` representing an open directory stream (returned by a call to `opendir()`).

Return Values

0 Success. The directory is closed.

-1 An error occurred. The directory file is not closed, and `errno` is set to indicate the error condition.

Description

The `closedir()` function closes the open directory file pointed to by *dirp*. The directory stream must be a structure of type `DIR` (defined in `<dirent.h>`) returned from a successful call to `opendir()`, or an error occurs. The file descriptor associated with the `DIR` structure is also closed.

Implementation Considerations

Refer to the `EFAULT` error description below.

The `DIR` structure is implemented using a file descriptor.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *dirp* parameter does not point to an open directory stream.
	ACTION	Pass a pointer to an open directory stream returned by the `opendir()` function.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *dirp* parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`opendir()`, `readdir()`, `rewinddir()`, `<dirent.h>`, POSIX.1 (Section 5.1.2)

confstr

Determine string-valued system configuration options.

Syntax

```
#include <unistd.h>
size_t confstr(int name, char *buf, size_t len);
```

Parameters

name Specifies the system configuration option, the string value of which you want to obtain. The value of name may be any one of a set of symbols defined in <unistd.h>; each of these symbols corresponds to a system configuration option. Possible symbols are:

_CS_PATH This name is used to return a value for the PATH environment variable that can find all the POSIX.2 standard utilities.

_CS_SHELL This name is used to find the path name to the standard shell command line interpreter.

buf Points to the region of memory where `confstr()` stores the string value of the variable indicated by name.

len Is the maximum number of characters that can be placed in buf. If this is not enough to hold the complete string value of name, `confstr()` truncates the string value to len-1 characters and appends a null terminator (the \0 character).

Return Values

The `confstr()` function returns configuration-defined string values.

If *name* is not a configuration defined value, then `confstr()` returns 0 and sets `errno`.

Description

`confstr()` is for options that have a string value; for options with a numeric value, use `sysconf()`. Unless there is an error, `confstr()` returns the length of the configuration defined string, including the null termination character. This may be greater than *len* if *len* wasn't big enough to hold the entire string value.

If *len* is zero and *buf* is a NULL pointer, `confstr()` does not attempt to return a string but does return the appropriate length. In this way, you can use the value to allocate sufficient memory to hold the string.

Errors

If an error occurs, `errno` is set to one of the following values:

`EINVAL`	CAUSE	The value specified for the name argument was invalid.
	ACTION	Specify a valid name.

If name has a configuration defined value, `confstr()` returns the size of the buffer required to hold that value. If this return value is greater than `len`, `confstr()` truncates the string returned in `buf`.

See Also

`sysconf()`, POSIX.2

creat

Creates a new file or rewrites an existing file.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (const char *pathname, mode_t mode);
```

Parameters

pathname A pointer to a string containing the pathname of a file to be created or rewritten. The pathname must be terminated by a null character.

mode File access permission bits. If the file already exists, *mode* is ignored. Access permission bits are set by ORing any combination of the following macros:

S_IRWXU	Set file owner class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRUSR	Set file owner class read permission bit.
S_IWUSR	Set file owner class write permission bit.
S_IXUSR	Set file owner class execute (if a file) or search (if a directory) permission bit.
S_IRWXG	Set file group class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRGRP	Set file group class read permission bit.
S_IWGRP	Set file group class write permission bit.
S_IXGRP	Set file group class execute (if a file) or search (if a directory) permission bit.
S_IRWXO	Set file other class read, write, and execute (if a file) or search (if a directory) permission bits.

S_IROTH	Set file other class read permission bit.
S_IWOTH	Set file other class write permission bit.
S_IXOTH	Set file other class execute (if a file) or search (if a directory) permission bit.

Return Values

>=0	Success. A nonnegative integer is returned representing the lowest numbered file descriptor not open by the calling process.
-1	An error occurred. The file is not opened, and `errno` is set to indicate the error condition.

Description

The `creat()` function opens for write-only access a file whose pathname is specified in the string pointed to by *pathname*.

The `creat()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file, and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to the file.

The `creat()` function returns a file descriptor for the specified file which is the lowest file descriptor not currently open for the calling process. The open file description is new; therefore, the file descriptor does not share it with any other process in the system.

If the file does not already exist, the file is created and the following occurs:

- The file offset is set to the beginning of the file.
- The file is opened for `O_WRONLY` access.
- The file's UID is set to the effective UID of the calling process.
- The file's GID is set to the GID of the directory in which the file is being created.
- The file permission bits of the file are set to *mode* and modified by the file mode creation mask of the calling process.
- The following file time fields are marked for update:
  - the file's `st_atime`, `st_ctime` and `st_mtime` time fields
  - the parent directory's `st_ctime` and `st_mtime` time fields

If the file already exists, the following occurs:

- The file is truncated to zero length, and the file offset is set to the beginning of the file.
- The file's UID, GID, and mode remain unchanged.
- The `st_ctime` and `st_mtime` time fields of the file are marked for update.

The function call

```
creat (path, mode);
```

is equivalent to

```
open (path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

Implementation Considerations

Refer to the `EACCES`, `EEXCL`, `EFAULT`, `EIMPL`, `EINVAL`, `EMFILE`, and `ESYSERR` error descriptions in the error section of the `open()` function description.

Pipes (or FIFOs), device special files, and read-only file systems are not supported through POSIX/iX interfaces and cannot be opened by `creat()`. Device files are inherited from the parent process, which has them opened as `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

The GID of a newly created file is set to the GID of the directory in which the file is created.

MPE/iX file equations are ignored by `creat()`.

The calling process must have the correct access permissions as defined by either an attached ACD or by the MPE/iX file security matrix. The calling process must have either ACD write access and append access or MPE/iX write access and append access.

Signals generated for the calling process during execution of `open()` are deferred from delivery until completion of this function.

Errors

Refer to the error section of the `open()` function description for errors returned by `creat()`. Possible errors returned by `creat()` are the same as those returned by `open()` when *oflag* is set to (`O_WRONLY | O_CREAT | O_TRUNC`).

See Also.

`open()`, `close()`, `dup()`, `execl()`, `execv()`, `<fcntl.h>`, `lseek()`, `read()`, `<signal.h>`, `fstat()`, `stat()`, `<stat.h>`, `write()`, `umask()`, POSIX.1 (Section 5.3.2)

ctermid

Identification of controlling terminal.

Syntax

```
#include <stdio.h>
char *ctermid(char *s);
```

Parameters

s The address of an array that will receive the pathname fo the current controlling terminal.

Return Values

If `s` is not NULL then it points to an array of *char L_ctermid* bytes long, or longer, as defined in <stdio.h>.

An empty string is returned if the `ctermid()` function is unsuccessful.

Description

The `ctermid()` function returns a string that can be used as a filename for referencing a terminal.

If a pathname is returned, access is not guaranteed.

Implementation Considerations

None.

Errors

None.

See Also

`ttyname()`, POSIX.1 (Section 4.7.2)

dup, dup2

Duplicates an open file descriptor.

Syntax

```
#include <unistd.h>
int dup (int fildes);

int dup2(iint fildes, int fildes2)
```

Parameters

fildes	An open file descriptor.

Return Values

>=0	Success. A new file descriptor is returned.
-1	An error occurred. the open file descriptor is not duplicated, and `errno` is set to indicate the error condition.

Description

The `dup()` and `dup2` functions return the lowest numbered file descriptor not currentely open by the calling process. The file descriptors returned by `dup()` and `dup2()` refer to the same open file description as *fildes* and share any locks.

The `dup()` and `dup2()` functions ignore file access permission bits when attempting to duplicate an open file descriptor.

Implementation Considerations

Refer to the `EEXCL` and `ESYSERR` error descriptions below.

Signals generated for the calling process during execution of `dup()` re deferred from delivery until completion of this function.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The parameter *fildes* is not a valid open file descriptor.
	ACTION	Check to see if the value passed in *fildes* has been altered or whether the file indicated by *fildes* was ever opened.
`EEXCL`	CAUSE	The specified file descriptor is opened for exclusive access.
	ACTION	Do not attempt to duplicate a file descriptor that is opened for exclusive access.
`EMFILE`	CAUSE	The number of open files and directories would exceed `{OPEN_MAX}`, the limit of opened files per process.
	ACTION	Check process limit in `<limits.h>`. Close a file.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`close()`, `creat()`, `execl()`, `execv()`, `open()`, POSIX.1 (Section 6.2.1)

execl

Executes a file.

Syntax

```
#include <unistd.h>
int execl (const char *pathname, const char *arg0, ...,
 const char *argn-1, (const char *)0);
```

Parameters

pathname A pointer to a string containing the pathname of the executable file that is to become the new process image. The pathname must be terminated by a null character.

The elements of the pathname must be uppercase and must resolve to a valid MPE/iX file, group, and account. For example, the pathname `/FINANCE/PAYROLL/JULY` must resolve to `JULY.PAYROLL.FINANCE`, where `JULY` is an executable file located in MPE/iX group `PAYROLL` in MPE/iX account `FINANCE`.

arg0, ..., argn Each of the parameters *arg0, ..., argn-1* point to a string containing an argument to the new process image. Each argument must be terminated by a null character. The last parameter, *argn*, must be a NULL pointer.

For an application to be strictly conforming, the first parameter, *arg0*, must point to a string containing a filename that identifies the executable file for the new process image.

Return Values

No return Success.

-1 An error occurred. The current process image remains unchanged, and `errno` is set to indicate the error condition.

Description

The `execl()` function replaces the current process image with a new process image created from the executable file specified in *pathname*.

Use the `execl()` function if you know the exact number of arguments to be passed to the new process image. Use the `execv()` function if the number of arguments passed to the new process image might vary at run time.

If the new process image is a C program, it is entered as a C function call having the following declaration:

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

In the above declaration, *argc* is a count of the number of pointers in the array *argv[]* and *argv[]* is an array of character pointers to the parameters *arg0* through *argn*. The NULL pointer terminating *argv[]* is not counted in *argc*.

The environment for the new process image is identical to the environment of the calling process.

If the new process image is not a C program, no information is made available through the argument list in *argv[]*.

The sum of the bytes used in both the argument list and environment list must not exceed `{ARG_MAX}` (defined in the file `<limits.h>`).

File descriptors open in the calling process image remain open in the new process image. For all file descriptors that remain open, all attributes of the open file description remain unchanged by this function, including file locks.

Streams open in the calling process image are not accessible in the new process image. (However, the underlying file descriptors that remain open, but inaccessible, are counted towards `{OPEN_MAX}`.)

Signals set to `SIG_DFL` or `SIG_IGN` in the calling process remain unchanged in the new process image. All signals of the calling process whose action is to invoke a signal handling function are set to `SIG_DFL` in the new process image.

The following attributes of the new process image are set to the same values of those of the calling process:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- time remaining until a SIGALRM signal
- current working directory
- root directory
- file mode creation mask
- process signal mask
- pending signals
- tms_utime, tms_stime, tms_cutime, and tms_cstime

The executable file's st_atime time field is marked for update. The executable file is open until the new process image terminates or executes another of the exec() functions.

Implementation Considerations

Refer to the EPERM, EIMPL, and ENOEXEC error descriptions below.

Some MPE/iX process attributes that are not specified in the POSIX 1003.1 standard are not inherited by the new process image.

NULL terminators and pointers are counted against {ARG_MAX}. Alignment bytes are counted against {ARG_MAX}.

The calling process's privilege level is used as the new program's maximum privilege level.

If the calling process entered debug mode through the DEBUG option of the MPE/iX CI RUN command, the new process image is also in debug mode.

Errors

If an error occurs, `errno` is set to one of the following values:

`E2BIG`	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of `{ARG_MAX}` (defined in `<limits.h>`).
	ACTION	Reduce the size of the argument list or environment list or both.
`EACCES`	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file. ■ The file is not a valid executable file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file. ■ Make sure that the file has an MPE/iX file code of `NMPRG`.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.

`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.

`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.

See Also

`execv()`, `fork()`, `alarm()`, `chmod()`, `_exit()`, `<signal.h>`, `sigprocmask()`, `sigpending()`, `fstat()`, `stat()`, `<sys/stat.h>`, `umask()`, POSIX.1 (Section 3.1.2)

execle

Executes a file.

Syntax

```
#include <unistd.h>
int execle (const char *path, const char *arg ...),
```

Parameters

path A pointer to a string containing the pathname of the executable file that is to become the new process image. The pathname must be terminated by a null character.

The elements of the pathname must be uppercase and must resolve to a valid MPE/iX file, group, and account. For example, the pathname `/FINANCE/PAYROLL/JULY` must resolve to `JULY.PAYROLL.FINANCE`, where `JULY` is an executable file located in MPE/iX group `PAYROLL` in MPE/iX account `FINANCE`.

arg0, ..., argn Each of the parameters *arg0, ..., argn-1* point to a string containing an argument to the new process image. Each argument must be terminated by a null character. The last parameter, *argn*, must be a NULL pointer.

For an application to be strictly conforming, the first parameter, *arg0*, must point to a string containing a filename that identifies the executable file for the new process image.

Return Values

No return Success.

-1 An error occurred. The current process image remains unchanged, and `errno` is set to indicate the error condition.

Description

The `execle()` function replaces the current process image with a new process image created from the executable file specified in *pathname*.

Use the `execl()` function if you know the exact number of arguments to be passed to the new process image. Use the `execv()` function if the number of arguments passed to the new process image might vary at run time.

If the new process image is a C program, it is entered as a C function call having the following declaration:

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

In the above declaration, *argc* is a count of the number of pointers in the array *argv[]* and *argv[]* is an array of character pointers to the parameters *arg0* through *argn*. The NULL pointer terminating *argv[]* is not counted in *argc*.

The environment for the new process image is identical to the environment of the calling process.

If the new process image is not a C program, no information is made available through the argument list in *argv[]*.

The sum of the bytes used in both the argument list and environment list must not exceed `{ARG_MAX}` (defined in the file `<limits.h>`).

File descriptors open in the calling process image remain open in the new process image. For all file descriptors that remain open, all attributes of the open file description remain unchanged by this function, including file locks.

Streams open in the calling process image are not accessible in the new process image. (However, the underlying file descriptors that remain open, but inaccessible, are counted towards `{OPEN_MAX}`.)

Signals set to `SIG_DFL` or `SIG_IGN` in the calling process remain unchanged in the new process image. All signals of the calling process whose action is to invoke a signal handling function are set to `SIG_DFL` in the new process image.

The following attributes of the new process image are set to the same values of those of the calling process:

- process ID
- parent process ID

- process group ID
- session membership
- real user ID
- real group ID
- time remaining until a SIGALRM signal
- current working directory
- root directory
- file mode creation mask
- process signal mask
- pending signals
- tms_utime, tms_stime, tms_cutime, and tms_cstime

The executable file's st_atime time field is marked for update. The executable file is open until the new process image terminates or executes another of the exec() functions.

Implementation Considerations

Refer to the EPERM, EIMPL, and ENOEXEC error descriptions below.

Some MPE/iX process attributes that are not specified in the POSIX 1003.1 standard are not inherited by the new process image.

NULL terminators and pointers are counted against {ARG_MAX}. Alignment bytes are counted against {ARG_MAX}.

The calling process's privilege level is used as the new program's maximum privilege level.

If the calling process entered debug mode through the ;DEBUG option of the MPE/iX CI RUN command, the new process image is also in debug mode.

Errors

If an error occurs, errno is set to one of the following values:

E2BIG

CAUSE: The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of {ARG_MAX} (defined in <limits.h>).

ACTION: Reduce the size of the argument list or environment list or both.

EACCES

CAUSE: One of the following:

- The calling process does not have search permission to a component of the pathname.
- The calling process does not have execute permission to the file.

ACTION: One of the following:

- Make sure that the calling process has search permission to all components of the pathname.
- Make sure that the calling process has execute permission to the file.

EFAULT

CAUSE: The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.

ACTION: Make sure that the pointer is correctly initialized.

EIMPL

CAUSE: The pathname did not resolve to a valid MPE/iX file, group, and account or the pathname begins with two slashes.

ACTION: Specify a valid pathname as described in the *pathname* parameter description.

ENAMETOOLONG

CAUSE: One of the following:

- The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>).
- A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.

ACTION: Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.

`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process in in a procedure exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.
`EBUSY`	CAUSE	The new process image file is loded by another user.
	ACTION	None.

See Also

`alarm()`, `chmod()`, `_exit()`, `fcntl()`, `fork()`, `setuid()`, <signal.h>, `sigpromask()`, `sigpending`, `stat()`, <sys/stat.h>, `times()`, `umask`, POSIX.1

execlp

Executes a file.

Syntax

```
#include <unistd.h> /* proto */
int execlp(const char *filename, const char *arg0, ...,);
```

Parameters

filename A pointer to a string containing the pathname of the executable file that is to become the new process image. The pathname must be terminated by a null character.

The elements of the pathname must be uppercase and must resolve to a valid MPE/iX file, group, and account. For example, the pathname `/FINANCE/PAYROLL/JULY` must resolve to `JULY.PAYROLL.FINANCE`, where `JULY` is an executable file located in MPE/iX group `PAYROLL` in MPE/iX account `FINANCE`.

arg0, ..., argn Each of the parameters *arg0, ..., argn-1* point to a string containing an argument to the new process image. Each argument must be terminated by a null character. The last parameter, *argn*, must be a NULL pointer.

For an application to be strictly conforming, the first parameter, *arg0*, must point to a string containing a filename that identifies the executable file for the new process image.

Return Values

No return Success.

-1 An error occurred. The current process image remains unchanged, and `errno` is set to indicate the error condition.

Description

The `execlp()` function replaces the current process image with a new process image created from the executable file specified in *filename*.

Use the `execlp()` function if you know the exact number of arguments to be passed to the new process image. Use the `execv()` function if the number of arguments passed to the new process image might vary at run time.

If the new process image is a C program, it is entered as a C function call having the following declaration:

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

In the above declaration, *argc* is a count of the number of pointers in the array *argv[]* and *argv[]* is an array of character pointers to the parameters *arg0* through *argn*. The NULL pointer terminating *argv[]* is not counted in *argc*.

The environment for the new process image is identical to the environment of the calling process.

If the new process image is not a C program, no information is made available through the argument list in *argv[]*.

The sum of the bytes used in both the argument list and environment list must not exceed `{ARG_MAX}` (defined in the file `<limits.h>`).

File descriptors open in the calling process image remain open in the new process image. For all file descriptors that remain open, all attributes of the open file description remain unchanged by this function, including file locks.

Streams open in the calling process image are not accessible in the new process image. (However, the underlying file descriptors that remain open, but inaccessible, are counted towards `{OPEN_MAX}`.)

Signals set to `SIG_DFL` or `SIG_IGN` in the calling process remain unchanged in the new process image. All signals of the calling process whose action is to invoke a signal handling function are set to `SIG_DFL` in the new process image.

The following attributes of the new process image are set to the same values of those of the calling process:

- process ID
- parent process ID

- process group ID
- session membership
- real user ID
- real group ID
- time remaining until a `SIGALRM` signal
- current working directory
- root directory
- file mode creation mask
- process signal mask
- pending signals
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

The executable file's `st_atime` time field is marked for update. The executable file is open until the new process image terminates or executes another of the `exec()` functions.

Implementation Considerations

Refer to the `EPERM`, `EIMPL`, and `ENOEXEC` error descriptions below.

Some MPE/iX process attributes that are not specified in the POSIX 1003.1 standard are not inherited by the new process image.

NULL terminators and pointers are counted against `{ARG_MAX}`. Alignment bytes are counted against `{ARG_MAX}`.

The calling process's privilege level is used as the new program's maximum privilege level.

If the calling process entered debug mode through the `;DEBUG` option of the MPE/iX CI `RUN` command, the new process image is also in debug mode.

Errors

If an error occurs, errno is set to one of the following values:

E2BIG	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of {ARG_MAX} (defined in <limits.h>).
	ACTION	Reduce the size of the argument list or environment list or both.
EACCES	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file.
EFAULT	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
EIMPL	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account or the pathname begins with two slashes.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.
ENAMETOOLONG	CAUSE	One of the following: ■ The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>). ■ A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.

`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.

See Also

`alarm()`, `chmod()`, `_exit()`, `fcntl()`, `fork()`, `setuid`, <signal.h>, `sigpromask()`, `sigpending`, `stat()`, <sys/stat.h>, `times()`, `umask()`, POSIX.1

execve

Executes a file.

Syntax

```
#include <unistd.h>
int execve (const char *path, char *const *argv[],
cr *const envp[]);
har *const envp[]);
```

Parameters

path A pointer to a string containing the pathname of the executable file that is to become the new process image. The pathname must be terminated by a null character.

The elements of the pathname must be uppercase and must resolve to a valid MPE/iX file, group, and account. For example, the pathname `/FINANCE/PAYROLL/JULY` must resolve to `JULY.PAYROLL.FINANCE`, where `JULY` is an executable file located in MPE/iX group `PAYROLL` in MPE/iX account `FINANCE`.

arg0, ..., argn Each of the parameters *arg0, ..., argn-1* point to a string containing an argument to the new process image. Each argument must be terminated by a null character. The last parameter, *argn*, must be a NULL pointer.

For an application to be strictly conforming, the first parameter, *arg0*, must point to a string containing a filename that identifies the executable file for the new process image.

envp[] An array of character pointers to null terminated strings. These strings constitute the environment for the new process image.

Return Values

No return	Success.
-1	An error occurred. The current process image remains unchanged, and `errno` is set to indicate the error condition.

Description

The `execve()` function replaces the current process image with a new process image created from the executable file specified in *pathname*.

Use the `execl()` function if you know the exact number of arguments to be passed to the new process image. Use the `execv()` function if the number of arguments passed to the new process image might vary at run time.

If the new process image is a C program, it is entered as a C function call having the following declaration:

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

In the above declaration, *argc* is a count of the number of pointers in the array *argv[]* and *argv[]* is an array of character pointers to the parameters *arg0* through *argn*. The NULL pointer terminating *argv[]* is not counted in *argc*.

The environment for the new process image is identical to the environment of the calling process.

If the new process image is not a C program, no information is made available through the argument list in *argv[]*.

The sum of the bytes used in both the argument list and environment list must not exceed `{ARG_MAX}` (defined in the file `<limits.h>`).

File descriptors open in the calling process image remain open in the new process image. For all file descriptors that remain open, all attributes of the open file description remain unchanged by this function, including file locks.

Streams open in the calling process image are not accessible in the new process image. (However, the underlying file descriptors that remain open, but inaccessible, are counted towards `{OPEN_MAX}`.)

Signals set to `SIG_DFL` or `SIG_IGN` in the calling process remain unchanged in the new process image. All signals of the calling process whose action is to invoke a signal handling function are set to `SIG_DFL` in the new process image.

The following attributes of the new process image are set to the same values of those of the calling process:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- time remaining until a `SIGALRM` signal
- current working directory
- root directory
- file mode creation mask
- process signal mask
- pending signals
- `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime`

The executable file's `st_atime` time field is marked for update. The executable file is open until the new process image terminates or executes another of the `exec()` functions.

Implementation Considerations

Refer to the `EPERM`, `EIMPL`, and `ENOEXEC` error descriptions below.

Some MPE/iX process attributes that are not specified in the POSIX 1003.1 standard are not inherited by the new process image.

NULL terminators and pointers are counted against `{ARG_MAX}`. Alignment bytes are counted against `{ARG_MAX}`.

The calling process's privilege level is used as the new program's maximum privilege level.

If the calling process entered debug mode through the `;DEBUG` option of the MPE/iX CI `RUN` command, the new process image is also in debug mode.

Errors

If an error occurs, `errno` is set to one of the following values:

`E2BIG`	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of `{ARG_MAX}` (defined in `<limits.h>`).
	ACTION	Reduce the size of the argument list or environment list or both.
`EACCES`	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account or the pathname began with two slashes.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.

`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.

`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.

See Also

`alarm()`, `chmod()`, `_exit()`, `fcntl()`, `fork()`, `setuid()`, <signal.h>, `sigpromask()`, `sigpending()`, `stat()`, <sys/stat.h>, `times()`, `umask`, POSIX.1

execvp

Executes a file.

Syntax

```
#include <unistd.h>
int execvp (const char *file, char * const argv[]);
```

Parameters

file A pointer to a string containing the pathname of the executable file that is to become the new process image. The pathname must be terminated by a null character.

The elements of the pathname must be uppercase and must resolve to a valid MPE/iX file, group, and account. For example, the pathname `/FINANCE/PAYROLL/JULY` must resolve to `JULY.PAYROLL.FINANCE`, where `JULY` is an executable file located in MPE/iX group `PAYROLL` in MPE/iX account `FINANCE`.

arg0, ..., argn Each of the parameters *arg0, ..., argn-1* point to a string containing an argument to the new process image. Each argument must be terminated by a null character. The last parameter, *argn*, must be a NULL pointer.

For an application to be strictly conforming, the first parameter, *arg0*, must point to a string containing a filename that identifies the executable file for the new process image.

Return Values

No return	Success.
-1	An error occurred. The current process image remains unchanged, and `errno` is set to indicate the error condition.

Description

The `execvp()` function replaces the current process image with a new process image created from the executable file specified in *pathname*.

Use the `execl()` function if you know the exact number of arguments to be passed to the new process image. Use the `execv()` function if the number of arguments passed to the new process image might vary at run time.

If the new process image is a C program, it is entered as a C function call having the following declaration:

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

In the above declaration, *argc* is a count of the number of pointers in the array *argv[]* and *argv[]* is an array of character pointers to the parameters *arg0* through *argn*. The NULL pointer terminating *argv[]* is not counted in *argc*.

The environment for the new process image is identical to the environment of the calling process.

If the new process image is not a C program, no information is made available through the argument list in *argv[]*.

The sum of the bytes used in both the argument list and environment list must not exceed `{ARG_MAX}` (defined in the file `<limits.h>`).

File descriptors open in the calling process image remain open in the new process image. For all file descriptors that remain open, all attributes of the open file description remain unchanged by this function, including file locks.

Streams open in the calling process image are not accessible in the new process image. The underlying file descriptors that remain open, but inaccessible, are counted towards `{OPEN_MAX}`.

Signals set to `SIG_DFL` or `SIG_IGN` in the calling process remain unchanged in the new process image. All signals of the calling process whose action is to invoke a signal handling function are set to `SIG_DFL` in the new process image.

The following attributes of the new process image are set to the same values of those of the calling process:

- process ID
- parent process ID

- process group ID
- session membership
- real user ID
- real group ID
- time remaining until a SIGALRM signal
- current working directory
- root directory
- file mode creation mask
- process signal mask
- pending signals
- tms_utime, tms_stime, tms_cutime, and tms_cstime

The executable file's st_atime time field is marked for update. The executable file is open until the new process image terminates or executes another of the exec() functions.

Implementation Considerations

Refer to the EPERM, EIMPL, and ENOEXEC error descriptions below.

Some MPE/iX process attributes that are not specified in the POSIX 1003.1 standard are not inherited by the new process image.

NULL terminators and pointers are counted against {ARG_MAX}. Alignment bytes are counted against {ARG_MAX}.

The calling process's privilege level is used as the new program's maximum privilege level.

If the calling process entered debug mode through the ;DEBUG option of the MPE/iX CI RUN command, the new process image is also in debug mode.

Errors

If an error occurs, errno is set to one of the following values:

E2BIG	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of {ARG_MAX} (defined in <limits.h>).
	ACTION	Reduce the size of the argument list or environment list or both.
EACCES	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file.
EFAULT	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
EIMPL	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account, or the pathname begins with two slashes.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.
ENAMETOOLONG	CAUSE	One of the following: ■ The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>). ■ A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.

`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.

See Also

`alarm()`, `chmod()`, `_exit()`, `fcntl()`, `fork()`, `setuid`, <signal.h>, `sigpromask()`, `sigpending`, `stat()`, <sys/stat.h>, `times()`, `umask`, POSIX.1

execv

Executes a file.

Syntax

```
#include <unistd.h>
int execv (const char *pathname, const char *argv[]);
```

Parameters

pathname A pointer to a string containing the pathname of the executable file that is to become the new process image. The pathname must be terminated by a null character.

The elements of the pathname must be uppercase and must resolve to a valid MPE/iX file, group, and account. For example, the pathname `/FINANCE/PAYROLL/JULY` must resolve to `JULY.PAYROLL.FINANCE`, where `JULY` is an executable file located in MPE/iX group `PAYROLL` and MPE/iX account `FINANCE`.

argv A pointer to an array where each element contains a pointer to a string containing an argument to the new process image. Each argument must be terminated by a null character. The element following the last element pointing to an argument must contain a NULL pointer.

For an application to be strictly conforming, the first element, *argv[0]*, must point to a string containing a filename that identifies the executable file for the new process image.

Return Values

No return Success.

-1 An error occurred. The current process image remains unchanged, and `errno` is set to indicate the error condition.

Description

The `execv()` function replaces the current process image with a new process image created from the executable file specified by *pathname*.

Use the `execl()` function if you know the exact number of arguments to be passed to the new process image. Use the `execv()` function if the number of arguments passed to the new process image might vary at run time.

If the new process image is a C program, it is entered as a C function call having the following declaration:

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

In the above declaration, *argc* is a count of the number of pointers in the array *argv[]*. The NULL pointer terminating *argv[]* is not counted in *argc*.

The environment for the new process image is identical to the environment of the calling process.

If the new process image is not a C program, no information is made available through the argument list in *argv[]*.

The sum of the bytes used in both the argument list and environment list must not exceed `{ARG_MAX}` (defined in the file `<limits.h>`).

File descriptors open in the calling process image remain open in the new process image. For all file descriptors that remain open, all attributes of the open file description remain unchanged by this function, including file locks.

Streams open in the calling process image are not accessible in the new process image. (However, the underlying file descriptors that remain open, but inaccessible, are counted towards `{OPEN_MAX}`.)

Signals set to `SIG_DFL` or `SIG_IGN` in the calling process remain unchanged in the new process image. All signals of the calling process whose action is to invoke a signal handling function are set to `SIG_DFL` in the new process image.

The following attributes of the new process image are set to the same values of those of the calling process:

- process ID
- parent process ID
- process group ID
- session membership
- real user ID
- real group ID
- time remaining until a SIGALRM signal
- current working directory
- root directory
- file mode creation mask
- process signal mask
- pending signals
- tms_utime, tms_stime, tms_cutime, and tms_cstime

The executable file's st_atime time field is marked for update. If the execv() function succeeds, the executable file is open until the new process image terminates or executes another of the exec() functions.

Implementation Considerations

Refer to the EPERM, EIMPL, and ENOEXEC error descriptions below.

Some MPE/iX process attributes that are not specified in the POSIX 1003.1 standard are not inherited by the new process image.

NULL terminators and pointers are counted against {ARG_MAX}. Alignment bytes are counted against {ARG_MAX}.

The calling process's privilege level is used as the new program's maximum privilege level.

If the calling process entered debug mode through the ;DEBUG option of the MPE/iX CI RUN command, the new process image is also in debug mode.

Errors

If an error occurs, errno is set to one of the following values:

E2BIG	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of {ARG_MAX} (defined in the file <limits.h>).
	ACTION	Reduce the size of the argument list or the environment list or both.
EACCES	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file. ■ The file is not a valid executable file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file. ■ Make sure that the file has an MPE/iX file code of NMPRG.
EFAULT	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
EIMPL	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.
ENAMETOOLONG	CAUSE	One of the following: ■ The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>). ■ A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.

`ENOENT`	CAUSE	A component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.

See Also

`execl()`, `fork()`, `alarm()`, `chmod()`, `_exit()`, `<signal.h>`, `sigprocmask()`, `sigpending()`, `fstat()`, `stat()`, `<sys/stat.h>`, `umask()`, POSIX.1 (Section 3.1.2)

_exit

Terminates a process.

Syntax

```
#include <unistd.h>
void _exit (int status);
```

Parameters

status A status code to be made available to the parent process of the calling process through the `wait()` or `waitpid()` functions.

Return Values

None. This function does not return to the calling process.

Description

The `_exit()` function terminates the calling process. The following actions are performed:

- The calling process is terminated.
- All open files and directory streams in the calling process are closed.
- The low-order 8 bits of the *status* parameter are saved and made available to the parent process through the `wait()` or `waitpid()` functions.
- All child processes of the calling process are terminated.
- A `SIGCHLD` signal is sent to the parent process to notify it of the calling process's termination.

Implementation Considerations

All child processes of the calling process are terminated. They are not adopted by a system process.

The CI session variable CJCW is set to *status*.

Time accounting information of the calling process is not made available to the parent process through the `wait()` or `waitpid()` functions. A zero is always returned.

No user process can be a controlling process. Only system processes (CI processes) are allowed to be controlling processes.

The controlling terminal is not disassociated from the session of the calling process.

Errors

None.

See Also

`close()`, `sigaction()`, `wait()`, `waitpid()`, POSIX.1 (Section 3.2.2)

fcntl

File control.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int *fildes, int cmd, ...);
```

Parameters

fildes An open file descriptor.

cmd The following values can be used for the file control command.

F_DUPFD	Duplicate file descriptor.
F_GETFD	Get file descriptor flags.
F_GETLK	Get record locking information.
F_SETFD	Set file descriptor flags.
F_GETFL	Get file status flags.
F_SETFL	Set file status flags.
F_SETLK	Set record locking information.
F_SETLKW	Set record locking information;wait if blocked.
FD_CLOEXEC	Close file descriptor upon execution of an *exec*-family function.

l_type Values for Record Locking With fcntl()

F_RDLCK	Shared or read lock.
F_UNLCK	Unlock.
F_WRLCK	Exclusive or write lock.

oflag Values for open()

O_CREAT	Create file if it does not exist.
O_EXCL	Exclusive use flag.
O_NOCITY	Do not assign a controlling terminal.
O_TRUNC	Truncate flag.

File Status Flags Used for open() and fcntl()

O_APPEND	Set append mode.
O_NONBLOCK	No delay.

File Access Modes Used for open() and fcntl()

O_RDONLY	Open for reading only.
O_RDWR	Open for reading and writing.
O_WRONLY	Open for writing only.

Mask for Use With File Access Modes

O_ACCMODE	Mask for file access modes.

Return Values

>=0	Success. A nonnegative integer is returned representing the lowest numbered file descriptor not open by the calling process.
-1	An error occurred. The file is not opened, and `errno` is set to indicate the error condition.

Description

Upon successful completion, the value returned will depend on *cmd*. The various return values are shown in Table 6-9.

Otherwise, a value of -1 will be returned and `errno` will be set to indicate the error.

The available values for *cmd* are defined in the <fcntl.h> (see 6.5.1) which will include:

F_DUPFD Return a new file descriptor that is the lowest numbered avaliable file descriptor greater than or equal to the third argument, *arg*, taken as an integer of type *int*. The new file descriptor refers to the same open file description as the original file descriptor and shares any locks.

The FD_CLOEXEC flag associated with the new file descriptor cleared to keep the file open across calls to the *exec* family of functions.

F_GETFD — Get the file descriptor flags that are associated with the file descriptor *fildes*. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

F_GETLK
F_SETFD — Set the file descriptor flags that are associated with *fildes* to the third argument, *arg* taken as type *int.* If the FD_CLOEXEC flag is zero, the file will be closed upon successful execution of an *exec* function.

F_GETFL — Get the file status flags and the file access modes for the open file description associated with *fildes.* The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in <fcntl.h>. File status flags and file access modes are associated with the open file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

F_SETFL — Set the file status flags for the open file description associated with *fildes* from the corresponding bits in the third argument, *arg*, taken as type *int*. Bits corresponding to the file access modes and the *oflag* values that are set in *arg* are ignored. If any bits in *arg* other than those mentioned here are changed by the application, the result is unspecified.

The following commands are available for advisory record locking. Advisory record locking shall be supported for regular files, and may be supported for other files.

F_GETLK — Get the first lock that blocks the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below) The information retrived overwrites the information passed to *fcntl()* in the *flock* structure. If no lock is found that would prevent this lock from being created, the structure will be left unchanged by this function call except for the lock type, which will be set to F_UNLCK.

F_SETLK
: Set or clear a file segment lock according to the lock description pointed to by the third argument, *arg*, taken as a pointer to type *struct flock* (see below). F_SETLK is used to establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks, (F_WRLCK), 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. If shared or exclusive lock cannot be set, *fcntl()* will return immediately.

F_SETLKW
: This command is the same as F_SETLK except that if a shared or exclusive lock is blocked by other locks, the process will wait until the request can be satisfied. If a signal that is to be caught is received while *fcntl* will be interrupted. Upon return from the signal handler of the process, *fcntl()* will return -1 with `errno` set to [EINTR], and the lock operation will not be done.

The *flock* structure, defined by the <fcntl.h> header, describes an advisory lock. It includes the members shown in Table 6-8.

When a shared lock has been set on a segment of a file, other processes will be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock will fail if the file descriptor was not opened with read access.

An exclusive lock will prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock will fail if the file descriptor was not opened with write access.

The value of *l_whence* is DEEK_SET, SEEK_CUR, or SEEK_END to indicate that the relative offset, *L_START* bytes, will be measured from the start of the file, current position, or end of the file, respectively, The value of *l_en* is the number of consecutive bytes to be locked. If *l_len* is negative, the result is undefined. The *l_pid* field is only used with F_GETLK to return the process ID of the process holeing a blocking lock. After a successful F_GETLK request, the value of *l_whence* will be SEEK_SET.

flock Structure

Member Type	Member Name	Description
short	*l_type*	F_RDLCK,F_WRLCK,or F_UNLCK.
short	*l_whence*	Flag for starting offset.
off_t	*l_start*	Relative offset in bytes.
off_t	*l_len*	Size;if 0,then until EOF.
pid_t	*l_pid*	Process ID of the process holding the lock, returned with F_GETLK.

Locks may start and extend beyond the current end of a file, but, will not start or extend before the beginning of the file. A lock will be set to extend to the largest possible value of the file offset for that file is *l_len* is set to zero. If the *flock struct* has *l_whence* and *l_start* that point to the beginning of the file, and *l_len* of zero, the intire file will be locked.

There will be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region will be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, and F_SETLK of an F_SETLKW request will (respectively) fail or block when another process has existing locks on bytes in the specified region and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process will be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process created useing the *fork()* function.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock the locked region of another process. If the system detects that sleeping until a locked region is unlocked would cause a deadlock, the *fcntl()* function will fail with an [EDEADLK]error.

Description

The function *fcntl()* provides for control over open files. The argument *fildes* is a file descriptor.

Implementation Considerations

The calling process must have the correct access permissions as defined by either an attached ACD or by the MPE/iX file security matrix. For example, a file opened `O_RDONLY` must have either ACD read access or MPE/iX read access. A file opened `O_WRONLY` or `O_RDWR` must have either ACD write access and append access or MPE/iX write access and append access.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The file does not exist and the calling process does not have write permission to the parent directory of the file to be created. ■ The file exists and the permissions specified by *oflag* are denied. ■ Both `O_TRUNC` and `O_RDONLY` were specified. ■ Both `O_APPEND` and `O_RDONLY` were specified. ■ An MPE/iX lockword is associated with the file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all directory components of the pathname. ■ Make sure that the calling process has write permission to the parent directory of the file to be created. ■ Specify valid and compatible flags in *oflag*. ■ Remove the MPE/iX lockword.

`EBADF`	CAUSE	The *fildes* argument is not a valid file descriptor. The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (l_type) is shared lock(F_RDLCK), and *fildes* is not a valid file descriptor open for reading. The argument *cmd* is F_SETLK or F_SETLKW, the type of lock (l_type) is an exclusive lock(F_WLRCK), and *fildes* is not a valid file descriptor open for writing.
	ACTION	None.
`EINTR`	CAUSE	The argument *cmd* is F_SETLKW, and the function was interrupted by a signal.
	ACTION	None.
`EINVAL`	CAUSE	More than one of the following three open flags were specified in *oflag*: `O_WRONLY`, `O_RDONLY`, and `O_RDWR`.
	ACTION	Specify only one of the open flags in *oflag*.
`EMFILE`	CAUSE	The argument *cmd* if F_DUPED and {OPEN_MAX} file descriptors are currently in use by this process, or no file descriptors greater than or equal to *arg* are available.
	ACTION	None.
`ENOLCK`	CAUSE	The argument *cmd* is F_SETLK or F_SETLKW, and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.
	ACTION	None.

For each of the following conditions, if the condition is detected, the *fcntl()* function will return -1 and set `errno` to the corresponding value:

`EDEADLK`	CAUSE	The argument *cmd* is F_SETLKW, and a deadlock condition was detected.
	ACTION	None.
`EAGAIN`	CAUSE	The system lacked the resources to create another process.
	ACTION	Attempt process creation at a later time, or decrease the number of processes associated with the application.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.

See Also

`close()`, `exec()`, `open()`, <fcntl.h>, (POSIX.1).

fnmatch

Compare filename to pattern (wild card) string.

Syntax

```
#include <fnmatch.h>
int fnmatch (const char *pattern, const char *string, int flags);
```

Parameters

pattern Is a string that may contain standard path name matching wild card characters. For example, asterisk (`*`), question mark (`?`), [] constructs, and so on.

string is a path name you want to compare to pattern.

flags specifies options for the match. Flags are represented by symbols defined in <fnmatch.h>. Recognized symbols are:

`FNM_NOESCAPE` disables backslash (\) escaping. When this flag is not set, the default behaviour is backslash escaping enabled; that is, if pattern contains a backslash (\) followed by a character, **`fnmatch()`** matches the character itself in string regardless of any special meaning it may have. For example, \\ in pattern matches \ in string.

`FNM_PATHNAME` indicates that slash (/) is a special character in string. For path names to match, pattern must have a slash wherever string does. For example, the string dir/file matches the pattern d* when FNM_PATHNAME is not given as a flag, but does not match when FNM_PATHNAME is present.

`FNM_PERIOD` indicates that a leading period in string must be matched by a period in pattern. An asterisk, question mark, or bracket expression does not match a leading period if FNM_PERIOD is set.

`FNM_IGNORECASE` indicates that case is to be ignored when comparing characters. For example, a matches A when this flag is set.

Return Values

0 string is a path name matching the wild card construct pattern.

`FNM_ERROR` error with the pattern and consequently no match.

`FNM_NOMATCH` there is no match.

Description

fnmatch() determines whether string is a path name matching the wild card construct pattern. If so, fnmatch() returns zero. If there is an error with the pattern and consequently no match, fmnatch() returns FNM_ERROR. If there is no match, fnmatch() returns the value FNM_NOMATCH.

Errors

None.

See Also

`regcomp()`, `regexec()`

fork

Creates a new child process.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

Parameters

None.

Return Values

>0	Success. The process ID of the newly created child process is returned to the calling process.
0	Success. A value of 0 is returned to the newly created child process.
-1	An error occurred. The process is not created, and `errno` is set to indicate the error condition.

Description

The `fork()` function creates a new child process. Both the new child process and the calling process (known as the parent process) continue execution upon the return from `fork()`. The new process is an exact copy of the calling process with the following exceptions:

- The child process has a unique process ID that does not match any active process group ID.
- The child process's parent process ID is that of the calling process.
- The child process has its own copy of the parent's file descriptors. Each of the child's file descriptors refers to the same open file description as the corresponding file descriptor of the parent.

- The child process has its own copy of the parent's open directory streams. Each open directory stream in the child process shares stream positioning with the corresponding directory stream of the parent.
- Directory streams are implemented using file descriptors. Both parent and child share the same open file descriptor for each directory stream.
- The child process's `tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` are set to zero.
- File locks set by the parent process are not inherited by the child process.
- Pending alarms are cleared for the child process.
- The set of signals pending for the child process is set to the empty set.

Implementation Considerations

Refer to the `EPERM` and `EIMPL` error descriptions below.

Some MPE/iX process characteristics not defined by POSIX are not inherited by the child. Examples are CM structures such as extra data segments, RINs, and SIRs.

The following MPE/iX characteristics not defined by POSIX.1 are inherited by the child:

- process's priority
- process's capability
- stack size
- heap size

Errors

If an error occurs, `errno` is set to one of the following values:

`EAGAIN`	CAUSE	The system lacked the resources to create another process.
	ACTION	Attempt process creation at a later time, or decrease the number of processes associated with the application.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the functional return argument.
	ACTION	Make sure that the functional return is correctly initialized.
`EIMPL`	CAUSE	The stack and heap could not be copied to the new process, or a file could not be inherited to the new process, or a system data structure could not be copied to the new process.
	ACTION	Contact your Hewlett-Packard Support Representative.
`ENOMEM`	CAUSE	The program requires more memory than the system allows for a process.
	ACTION	Reduce memory requirements for the process.

`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process is not executing a program file whose MPE/iX file code is `NMPRG`. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Make sure that the calling process has the MPE/iX PH capability. ■ Make sure that the calling process is executing a program file whose file code is `NMPRG`. ■ Do not execute `fork()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.

See Also

`alarm()`, `execl()`, `execv()`, `kill()`, `wait()`, POSIX.1 (Section 3.1.1)

fpathconf

Returns configuration variable for file descriptor.

Syntax

```
#include <unistd.h>
long fpathconf(int fildes, int name);
```

Parameters

fildes	is an open file descriptor for the file or directory of which you want to determine the configuration variables.
name	is a symbol indicating the variable, the value of which you want to determine.

Return Values

variable value	`fpathconf()` lets you determine the value of a configuration variable associated with a particular file descriptor. If `fpathconf()` can determine the value of the requested variable, it returns that value as its result.
-1	If `fpathconf()` cannot determine the value of the specified variable, it returns -1 and sets errno

`fpathconf()` works exactly like `pathconf()`, except that it takes a file descriptor as an argument rather than a path name. For further details, see `pathconf()`.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	*fildes* was not a valid file descriptor.
	ACTION	Specify a valid file descriptor.
`EINVAL`	CAUSE	Name was not a valid variable code, or the given variable cannot be associated with the specified file.
	ACTION	Specify a valid variable code.

See Also

`pathconf()`

fstat

Returns open file status information.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
int fstat (int fildes, struct stat *buffer);
```

Parameters

fildes An open file descriptor.

buffer A pointer to a buffer of type `struct stat` (defined in `<sys/stat.h>`) where file information is returned.

Return Values

0 Success.

-1 An error occurred. File status information is not returned and `errno` is set to indicate the error condition.

Description

The `fstat()` function returns status information on the open file specified by *fildes*. In order to use `fstat()` on an open directory stream, the directory stream associated with the open directory must be converted to a file descriptor by calling the ANSI C function `fileno()`.

The `fstat()` function updates to the current time all time fields that have been previously marked for update. All update marks are removed.

Implementation Considerations

Refer to the `EFAULT`, `EPERM`, and `ESYSERR` error descriptions below.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *fildes* parameter is not a valid open file descriptor.
	ACTION	Pass a valid open file descriptor.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *buffer* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process is not executing a program file whose MPE/iX file code is `NMPRG`. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Make sure that the calling process has the MPE/iX PH capability. ■ Make sure that the calling process is executing a program file whose file code is `NMPRG`. ■ Do not execute `fork()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
`ESYSERR`	CAUSE	Access denied. Unable to map UID and GID to owner of the file or directory designated by *fildes*, either because user database is corrupted or file is invalid for the POSIX/iX environment.
	ACTION	Check user database or if access to the file is valid in the POSIX/iX environment.

See Also

`creat()`, `dup()`, `open()`, `<sys/stat.h>`, POSIX.1 (Section 5.6.2)

getcwd

Returns the pathname of the current working directory.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
char *getcwd (char *buffer, size_t size);
```

Parameters

buffer A pointer to a character array where an absolute pathname for the calling process's current working directory is returned. The pathname must be terminated by a null character. The size of the array must be large enough to contain the length of the pathname plus the terminating null character.

size The size, in bytes, of the array pointed to by *buffer*.

Return Values

<>NULL Success. A pointer to *buffer* is returned.

NULL An error occurred. The contents of *buffer* are undefined, and `errno` is set to indicate the error condition.

Description

The `getcwd()` function places in the array pointed to by *buffer* the absolute pathname of the calling process's current working directory. Any contents of *buffer* past the terminating null character are undefined. If an error occurs, the contents of the *buffer* are undefined.

Implementation Considerations

Refer to the `EFAULT` and `ESYSERR` error descriptions below.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	The calling process either does not have search permission to a component of the pathname or does not have read permission to the current working directory.
	ACTION	Make sure that the calling process has search permission to all component directories in the pathname and read permission to the current working directory.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *buffer* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINVAL`	CAUSE	The *size* parameter is equal to zero.
	ACTION	Make sure that the *size* parameter is greater than zero.
`ERANGE`	CAUSE	The *size* parameter specifies a length that is less than the size of the current working directory pathname plus one.
	ACTION	Pass enough buffer area to contain a full pathname.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`chdir()`, POSIX.1 (Section 5.2.2)

getegid

Returns the effective group ID.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
gid_t getegid (void);
```

Parameters

None.

Return Values

The effective group ID of the calling process.

Description

The `getegid()` function returns the effective group ID (GID) of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`getgid()`, `geteuid()`, `getuid()`, POSIX.1 (Section 4.2.1)

getenv

Returns an environment value.

Syntax

```
#include <stdlib.h>
char *getenv (const char *name);
```

Parameters

name	A pointer to a string of characters to match in the environment list.

Return Values

<>NULL	A pointer to the *value* portion of a `name=`*value* string is returned.
NULL	A matching name was not found, and `errno` is not modified.
NULL + `errno`	An error occurred, and `errno` is set to indicate the error condition.

Description

The `getenv()` function takes a string, *name*, and searches for a matching name in the environment list (in `environ`) associated with the calling process.

If a match to *name* is found, `getenv()` returns a pointer to the *value* portion of that string. The *value* is terminated by a null character. If a matching name is not found, `getenv()` returns a NULL pointer but does not modify the current value of `errno`.

The environment list contains strings in the form `name=`*value*. If more than one string has the same name, `getenv()` returns the value for the first matching name found. The length of *name* is limited by `{ARG_MAX}` as defined in `<limits.h>`.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to the following value:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *name* parameter or while dereferencing `environ` and traversing the process's environment list.
	ACTION	Check to see if the pointer is correctly initialized or if the environment list is corrupted.

See Also

`environ()`, POSIX.1 (Section 4.6.1)

geteuid

Returns the effective user ID.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
uid_t geteuid (void);
```

Parameters

None.

Return Values

The effective UID of the calling process.

Description

The `geteuid()` function returns the effective user ID (UID) of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`getegid()`, `getgid()`, `getuid()`, POSIX.1 (Section 4.2.1)

getgid

Returns the real group ID.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
gid_t getgid (void);
```

Parameters

None.

Return Values

The real GID of the calling process.

Description

The `getgid()` function returns the real group ID (GID) of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`getegid()`, `geteuid()`, `getuid()`, POSIX.1 (Section 4.2.1)

getgrgid

Group data base access based on GID.

Syntax

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrgid(gid_t gid);
```

Parameters

gid A value of a GID.

Return Values

Returns a pointer to an object of type struct group on success. The return values may point to static data that is overwritten by each cell.

A null pointer is returned on error or if the requested entry is not found.

Description

The *getgrgid()* function returns a pointer to an object of type *struct group* containing an entry from the group database with a matching GID. This structure, which is defined in <grp.h>, includes the members shown below:

gr_name The name of the group.

gr_gid The numerical group ID.

gr_mem A null-terminated vector of pointers to the individual member names.

Implementation Considerations

Currently, member gr-mem has not been implemented. It returns NULL.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address when attempting to allocate or access a struct group buffer.
	ACTION	Report circumstances to HP
`EINVAL`	CAUSE	The GID parameter is invalid. No matching entry was found in the group database.
	ACTION	Specify a valid GID.
`ESYSERR`	CAUSE	The system detected an unexpected error.
	ACTION	Report circumstances to HP.

See Also

`getlogin()`,`getgrnam()` POSIX.1

getgrnam

Group data base access.

Syntax

```
#include <sys/types.h>
#include <grp.h>
struct group *getgrnam(const char *name);
```

Parameters

name A character-string value.

Return Values

Returns a pointer to an object of type struct group on success. The return values may point to static data that is overwritten by each cell.

A null pointer is returned on error or if the requested entry is not found.

Description

The *getgrnam()* routine returns a pointer to an object of type *struct group* containing an entry from the group database with a matching *name*. This structure, which is defined in <grp.h>, includes the members shown below:

gr_name The name of the group.

gr_gid The numerical group ID.

gr_mem A null-terminated vector of pointers to the individual member names.

Implementation Considerations

Currently, member gr-nam has not been implemented. It returns NULL.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address when attempting to allocate or access a struct group buffer.
	ACTION	Report circumstances to HP
`EINVAL`	CAUSE	The name is invalid. No matching entry was found in the group database.
	ACTION	Specify a valid name.
`ESYSERR`	CAUSE	The system detected an unexpected error.
	ACTION	Report circumstances to HP.

See Also

`getlogin()`, POSIX.1

getgroups

Gets Supplementary Group IDs.

Syntax

```
#include <sys/types.h>
int getgroups (int *gidsetsize, gid_t grouplist[]);
```

Parameters

gidsetsize The number of elements in the *grouplist* array.

grouplist An array containing the supplementary group IDs of the calling process.

Return Values

Upon successful completion, the number of supplementary group IDs is returned. This value is zero if {NGROUPS_MAX} is zero. A return value of -1 indicates failure, and `errno` is set to indicate the error.

Description

The *getgroups()* function fills in the array *grouplist* with the supplementary group IDs of the calling process. The *gidsetsize* argument specifies the number of elements in the supplied *grouplist* array. The actual number of suplementary group IDs stored in the array is returned. The values of array entries with indices larger than or equal to the returned value are undefined.

As a special case, if the *gidsetsize* argument is zero, *getgroups()* returns the number of supplemental group IDs associated with the calling process without modifying the array pointed to by the *grouplist* argument.

Implementation Considerations

Supplemental group IDs are not currently supported ({NGROUPS_MAX} is 0). Therefore, this function will always return 0.

Errors

If an error occurs, `errno` is set to one of the following values:

`EINVAL`	CAUSE	*gidsetsize* is not equal to zero and is less than the number of supplemental group IDs.
	ACTION	Specify a valid and supported value.

See Also

`getgid()`, POSIX.1

getlogin

Gets user name.

Syntax

```
#include <unistd.h>
char *getlogin(void);
```

Parameters

None.

Return Values

Returns a pointer to a string on success. The return values may point to static data that is overwritten by each cell.

A null pointer is returned on error or if the user's login name cannot be found.

Description

The `getlogin()` function returns a pointer to a string giving a user name associated with the calling process.

Implementation Considerations

The users login name string will be in the form "USER.ACCOUNT".

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to allocate or access a string buffer area in which to move the user's login name.
	ACTION	Report circumstances to HP.
`ESYSERR`	CAUSE	The system detected an unexpected error.
	ACTION	Report circumstances to HP.

See Also

`getpwnam()`, `getpwuid`, POSIX.1

getopt

Command option parsing.

Syntax

```
#include <unistd.h>
extern char *optarg;
extern int optind, opterr, optopt;
int getopt(int argc, const char *argv[],
           const char *optstring);
```

Parameters

argc is the argument count as passed to `main()`.

argv[] is the argument vector as passed to `main()`.

optstring is a string containing letters and/or digits which should be recognized as command line arguments. For example, if a program takes the arguments -a, -A, and -b, optstring could be "aAb". The characters in optstring may be in any order. If an option may take an argument, the option character in optstring should be followed by a colon. For example, if the example command also takes an option

`-c` *value*

optstring could be "aAbc:". A colon as the first character of *optstring* returns a : (instead of a ?) if `getopt` encounters a missing argument.

Return Values

-1 When it reaches the end of the options. This can be when argv[optind] is NULL or the strings - or—, or when a command line argument does not begin with -. If argv[optind] is—, getopt() increments optind by 1; otherwise, it does not increment optind.

? If getopt() encounters an invalid option (one whose character does not appear in optstring) or an option that was supposed to be followed by an argument value but was not, getopt() returns a question mark (?). If the first character is a :, and the error is a missing argument, then : is returned instead of ?. The character that caused the error is assigned to the variable optopt, and optind is not updated.

1 Normally, getopt() writes an error message to the standard error stream if it encounters an error; to disable this error message, assign the value zero to the variable opterr or start the optstring with :. By default, opterr is initialized to 1.

Description

`getopt()` helps parse a command line that corresponds to the standard POSIX.2 syntax: options are single letters or digits marked with a minus (-) and possibly followed by a value. `getopt()` recognizes that options may be concatenated; for example,

```
-a -b
```

can be combined into

```
-ab
```

`getopt()` returns the character that represents the option. For example, if `getopt()` identifies the -a option, it returns 'a'.

Successive calls to `getopt()` obtain successive options from the command line. `getopt()` uses the variable optind to keep track of which argv element it examines . optind is initialized to 1, and every invocation of `getopt()` sets optind to the command line argument to be scanned. When a single argument contains several options (as in -abc), optind indicates the same *argv* element until all the options have been returned.

If an option takes an argument, `getopt()` sets *optarg* to point to the associated argument, according to these rules:

*If the option character was at the end of an *argv* element, the associated argument is assumed to be the element of argv. In this case, `optind` is incremented by 2; otherwise, the argument value is assumed to come immediately after the argument letter. It is the rest of the argv element. In this case, optind is incremented by 1.

Example

The following code fragment shows how one might process the arguments for a utility that can take the mutually exclusive options *a* and *b* and the options *f* and *o*, both of which require arguments.

```
#include <unistd.h>

int main (int argc, char *argv[])
{
int c, bflg, aflg, errflg = 0;
char *ifile, *ofile;
extern char *optarg;
extern int optind, optopt;
. . .
while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
switch (c) {
case 'a':
if (bflg)
errflg = 1;
else
aflg = 1;
break;
case 'b':
if (aflg)
errflg = 1;
else
bflg = 1;
bproc( );
break;
case 'f':
ifile = optarg;
break;
case 'o':
ofile = optarg;
break;
```

```
case ':': /* -f or -o without option-arg */
fprintf (stderr,
"Option -%c requires an option-argument0,
optopt);
errflg = 1;
break;
case '?':
fprintf (stderr,
"Unrecognized option: -%c0, optopt);
errflg = 1;
break;
}
}
if (errflg) {
fprintf(stderr, "usage: . . . ");
exit(2);
}
for ( ; optind < argc; optind++) {

if (access(argv[optind], R_OK)) {
. . .
}
```

Errors

If an error occurs, `errno` is set to one of the following values:

Option -option argument missing	CAUSE	When invoking a program that calls getopt(), you specified -option but did not provide the argument that optstring indicated.
	ACTION	Provide the missing argument.
Unknown option "-option"	CAUSE	When invoking a program that calls getopt(), you specified an option that was not in optstring.
	ACTION	Specify an option included in optstring.

Implementation Considerations

The current implementation of MPE/iX uses the INFO string to pass arguments to programs. If the size of this string plus the size of the current environment (determined by the number and size of the environment variables in the current process) is greater than 8192 bytes, the string is too long to pass to a subprocess and the process creation fails.

See Also

`getopt(1)`, `getopts(1)`

getpid

Returns the process identification number.

Note If linking with the POSIX/iX libraries, refer to the description of `getpid()` located in the *MPE/iX Developer's Kit Reference Manual*.

Syntax

```
int getpid (void)
```

Parameters

None.

Return Values

x	The process identification number (PIN) of the calling process.

Implementation Considerations

None.

See Also

MPE/iX intrinsics `FATHER` and `GETPROCID`, described in the *MPE/iX Intrinsics Reference Manual*.

getpwuid

User database access based on UID.

Syntax

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

Parameters

uid A value of a user ID.

Return Values

Returns a pointer to an object of type struct passwd on success. The return values may point to static data that is overwritten by each cell.

A null pointer is returned on error or if the requested entry is not found.

Description

The *getpwuid()* function returns a pointer to an object of type *struct passwd* containing an entry from the group database with a matching *uid* or *name*. This structure, which is defined in <pwd.h>, includes the members shown in the following:

pw_name User name

pw_uid User ID number

pw_gid Group ID number

pw_dir Initial working directory

pw_shell Initial User Program

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address when attempting to allocate or access a struct passwd buffer.
	ACTION	Report circumstances to HP
`EINVAL`	CAUSE	the UID parameter is invalid. No matching entry was found in the passwd buffer.
	ACTION	Specify a valid UID.
`ESYSERR`	CAUSE	The system detected an unexpected error.
	ACTION	Report circumstances to HP.

See Also

`getlogin()`, `getpwnam()` POSIX.1

getpgrp

Returns the process group ID.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp (void);
```

Parameters

None.

Return Values

The process group ID of the calling process.

Description

The `getpgrp()` function returns the process group ID of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`getpid()`, `sigaction()`, POSIX.1 (Section 4.3.1)

getpid

Returns the process ID.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid (void);
```

Parameters

None.

Return Values

The process ID of the calling process.

Description

The `getpid()` function returns the process ID (PID) of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`getppid()`, `execl()`, `execv()`, `fork()`, `kill()`, POSIX.1 (Section 4.1.1)

getppid

Returns the parent's process ID.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
pid_t getppid (void);
```

Parameters

None.

Return Values

The parent process ID of the calling process.

Description

The `getppid()` function returns the parent process ID of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`getpid()`, `execl()`, `execv()`, `fork()`, `kill()`, POSIX.1 (Section 4.1.1)

getpwnam

User database access based on UID

User database access.

Syntax

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwnam(const char *name);
```

Parameters

name A character string value corresponding to the user name.

Return Values

Returns a pointer to an object of type struct *passwd* on success. The return values may point to static data that is overwritten by each cell.

A null pointer is returned on error or if the requested entry is not found.

Description

The *getpwnam()* function is used to obtain entry from the user database with a matching name. This structure, which is defined in <pwd.h>, includes the members shown below:

pw_name User name

pw_uid User ID number

pw_gid Group ID number

pw_dir Initial working directory

pw_shell Initial User Program

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address when attempting to allocate or access a struct passwd buffer.
	ACTION	Report circumstances to HP
`EINVAL`	CAUSE	The name parameter is invalid. No matching entry was found in the group database.
	ACTION	Specify a valid name.
`ESYSERR`	CAUSE	The system detected an unexpected error.
	ACTION	Report circumstances to HP.

See Also

`getlogin()`, `getpwuid()`, POSIX.1

getpwuid

User database access based on UID.

Syntax

```
#include <sys/types.h>
#include <pwd.h>
struct passwd *getpwuid(uid_t uid);
```

Parameters

uid A value of a user ID.

Return Values

Returns a pointer to an object of type struct passwd on success. The return values may point to static data that is overwritten by each cell.

A null pointer is returned on error or if the requested entry is not found.

Description

The *getpwuid()* function returns a pointer to an object of type *struct passwd* containing an entry from the group database with a matching *uid*. This structure, which is defined in <pwd.h>, includes the members shown in the following:

pw_name User name

pw_uid User ID number

pw_gid Group ID number

pw_dir Initial working directory

pw_shell Initial User Program

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to one of the following values:

`E2BIG`	CAUSE	The number of bytes used by the new process image's argument list and environment list combined is greater than the limit of `{ARG_MAX}` (defined in `<limits.h>`).
	ACTION	Reduce the size of the argument list or environment list or both.
`EACCES`	CAUSE	One of the following: ▪ The calling process does not have search permission to a component of the pathname. ▪ The calling process does not have execute permission to the file.
	ACTION	One of the following: ▪ Make sure that the calling process has search permission to all components of the pathname. ▪ Make sure that the calling process has execute permission to the file.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.

`EIMPL`	CAUSE	The pathname did not resolve to a valid MPE/iX file, group, and account, or the pathname begins with two slashes.
	ACTION	Specify a valid pathname as described in the *pathname* parameter description.
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOEXEC`	CAUSE	The program file does not have the `NMPRG` file code.
	ACTION	Make sure that the program file has the `NMPRG` file code.
`ENOMEM`	CAUSE	The new process image requires more memory than the system allows.
	ACTION	No action required. The new process image cannot be created.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.

`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.
`ESYSERR`	CAUSE	System error occurred when accessing a system database.
	ACTION	None.

See Also

`getlogin()`, `getpwnam()`, POSIX.1

getuid

Returns the real user ID (UID).

Syntax

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid (void);
```

Parameters

None.

Return Values

The real UID of the calling process.

Description

The `getuid()` function returns the real user ID (UID) of the calling process.

Implementation Considerations

None.

Errors

None.

See Also

`geteuid()`, `getegid()`, `getgid()`, POSIX.1 (Section 4.2.1)

glob

Generate path name list matching pattern.

Syntax

```
#include <glob.h>
int glob(const char *pattern, int flags,
 int (*errfunc)(const char *name, int errno),
 glob_t *paths);
```

Parameters

pattern is a string giving a path name pattern, possibly containing wild card characters and other path name generation constructs.

flags is a collection of flags controlling the `glob()` action. Flags are specified by ORing together symbolic constants defined in <glob.h>. Possible symbols are:

GLOB_APPEND appends path names to an existing paths list generated by a previous call to `glob()`.

GLOB_DOOFFS uses the `gl_offs` field in the `glob_t` structure paths.

GLOB_ERR tells `glob()` to return when it encounters a directory it cannot open or read. By default, `glob()` continues to look for matches (see also errfunc.)

GLOB_MARK distinguishes between directories and files with names that match pattern

	by putting a slash (/) after directory names.
GLOB_NOCHECK	takes a special action if no path names match pattern. By default, `glob()` returns a null list if there are no path names matching pattern. However, if GLOB_NOCHECK is specified, `glob()` returns a list consisting only of pattern and indicates that the number of matched path names is 1. You might use this option if an argument can be either a path name or a normal string.
GLOB_NOESCAPE	turns off escape character functionality. By default, `glob()` treats a backslash (\) as the escape character.
GLOB_NOSORT	does not sort matching path names. By default, `glob()` sorts the path names according to the current locale's collating sequence.

errfunc is a function to be called if `glob()` finds a directory that cannot be opened or read. If the errfunc pointer is NULL, `glob()` ignores such directories; otherwise, `glob()` calls the function indicated by errfunc, passing two arguments:

- a *const* char * giving the name of the directory that could not be opened or read;

- an `int` giving the value of `errno` set by the function that tried to open or read the directory. This function could be `opendir()`, `readdir()`, or `stat()`.

paths points to an area where `glob()` can store a `glob_t` structure. This structure gives the list of path names matching pattern and other information. It must be created by the caller.

Description

`glob()` generates a list of all accessible path names matching *pattern*. For access to a path name, `glob()` must have search permission on every component of the path name except the last, and must have read permission on the parent directory of each filename component of pattern that contains any of the wild card characters *, ?, or [.

The path name list is given using a `glob_t` structure. This structure has the following fields:

size_t gl_pathc
: is the number of path names that match pattern. This is zero if `glob()` finds no matching path names. However, if GLOB_NOCHECK is specified, gl_pathc is always 1, as discussed under the description of GLOB_NOCHECK in the Parameters section. This field is set by `glob()`.

char **gl_pathv
: points to a list of strings giving the path names that matched pattern. The first pointer after the last path name is NULL. This field is set by `glob()`.

size_t gl_offs
: tells how many NULL pointers you want at the beginning of the gl_pathv list. This creates a specified amount of blank space at the beginning of gl_pathv that can be used for other purposes. For example, you might fill this space with other arguments before passing the whole gl_pathv vector as an argument to a function like `execv()`.

Before calling `glob()`, set gl_offs to the number of NULL pointers that `glob()` inserts in the gl_pathv list. These NULL pointers precede the pointers to the strings which identify path names that match pattern. `glob()` only uses the value in gl_offs if you have set GLOB_DOOFFS in flags.

If GLOB_APPEND is specified to add new path names to an existing list, `glob()` follows these rules:

- If GLOB_DOOFFS is set in the first call to `glob()`, it must be set in subsequent calls and gl_offs must have the same value in each call;
- If GLOB_DOOFFS is not set in the first call, it must not be set in subsequent calls;
- After the second call, gl_pathv points to a list containing:
  - The number of NULL pointers as determined by GLOB_DOOFFS and gl_offs;
  - Pointers to the path names that were in the list before the second call, in the same order as before;
  - Pointers to the new path names obtained by the second call, in the order dictated by the flags for the second call.
- gl_pathc gives the total number of path names from all the calls.

The application should not change gl_pathc or gl_pathv between calls.

As noted earlier, the function given by (*`errfunc`) () is called if `glob()` encounters a directory that cannot be opened or read. If (*`errfunc`) () returns non-zero or if GLOB_ERR is set in flags, `glob()` sets paths to reflect the path names already obtained, then returns with a result of GLOB_ABORTED. (This symbolic constant is defined in <glob.h>.) If GLOB_ERR is not specified and if either errfunc is NULL or (*`errfunc`) () returns zero, `glob()` ignores the error and continues searching for matching path names.

Return Values

0	Completes successfully
Error Value	Not successful

If `glob()` terminates prematurely with one of these errors, it still sets paths->gl_pathc and paths->gl_pathv to show whatever path names it has already found.

Once a program has finished using the paths structure, it should use `globfree()` to free up the space used to store the path name list.

Errors

If an error occurs, `errno` is set to one of the following values:

`GLOB_NOSPACE`	CAUSE	`glob()` was unable to allocate memory for at least one of the path names obtained.
	ACTION	Free up more memory.
`GLOB_NOMATCH`	CAUSE	`glob()` did not find any path names which matched pattern and GLOB_NOCHECK was not set in the flags.
	ACTION	No action required.
`GLOB_ABORTED`	CAUSE	`glob()` stopped because GLOB_ERR was set (perhaps a directory could not be opened or read), or because (`*errfunc) ()` returned non-zero.
	ACTION	Check that the offending directory exists, that it was named properly, and that you have appropriate permissions.

See Also

`fnmatch()`, `globfree()`

globfree

Release data created by `glob()`.

Syntax

```
#include <glob.h>
void globfree(glob_t *paths);
```

Parameters

paths is a `glob_t` structure used in a previous call to `glob()`.

Description

`globfree()` frees any memory allocated in connection with the paths structure. Typically, this gets rid of any space that a call to `glob()` allocated to hold a path name list.

Errors

None.

See Also

`glob()`

ioctl-mag_tape

Provides an interface and control over magnetic tape devices. In the case of magnetic tape devices, the `ioctl()` function provides an interface for issuing various control commands to opened tape devices. The `ioctl()` operations can be used to position the magnetic tape, and to determine the tape device status.

Syntax

```
int ioctl (fildes, request, arg)
int fildes;
int request;
void *arg;
```

Parameters

fildes
: The file descriptor of the successfully opened device.

request
: For magnetic tape devices, this parameter specifies which type of command to perform. In addition to the desired command, the *request* parameter is made up of several fields which encode the size and direction of the *arg* parameter. The two types of *requests* that are available are described below.

  `MTIOCTOP` This *request* is used to position the magnetic tape device.

  `MTIOCGET` This *request* is used to retrieve the magnetic tape device status.

  The `MTIOCTOP` and `MTIOCGET` requests are defined in <sys/mtio.h>.

arg
: Depending on the type of *request* specified, the *arg* parameter will be equal to one of the structure listed below. The following structures are defined in <sys/mtio.h>.

Return Values

0 The function completed successfully.

-1 An error occured. The value of -1 is returned by the function, and the global variable `errno` is set with the resultant error.

Description

If the *request* is `MTIOCTOP`, then the *arg* parameter will be equated to the following structure:

```
full
struct mtop  {
       short mt_op;    /* operations to be performed */
       long  mt_count; /* number of times to performed */
                       /* the specified operation */
            };
```

The different operations that can be performed are as follows.

`MTWEOF` Writes an end of file record.

`MTFSF` Moves the tape forward until a tape mark is encountered.

`MTBSF` Moves the tape backward until a tape mark is encountered.

`MTFSR` Moves the tape forward a specified number of records.

`MTBSR` Moves the tape backward a specified number of records.

`MTREW` Rewinds the tape.

`MTOFFL` Rewinds the tape and puts the drive offline.

`MTNOP` This operation is not supported.

`MTEOD` This operation is not supported.

`MTWSS` For DDS devices only. Writes and saves the setmark.

`MTFSS` For DDS devices only. Spaces forward to the setmark.

`MTBSS` For DDS devices only. Spaces backward to the setmark.

If the *request* is `MTIOCGET`, then the *arg* parameter returned will be equated to the following structure:

```
full
struct mtget  {
       long   mt_type;   /* type and subtype of device */
       long   mt_resid;  /* not supported */
       long   mt_dsreg1; /* not supported */
       long   mt_dsreg2; /* not supported */
       long   mt_gstat;  /* generic device status */
       long   mt_erreg;  /* not supported */
       long   mt_fileno; /* not supported */
       long   mt_blkno;  /* not supported */
              };
```

The `mt_type` that is returned will be a integer with the upper 16 bits representing the device type, and the lower 16 bits representing the device subtype. The device type and subtype, will be returned in the hexadecimal format listed below:

```
MT_7976      0x180001    /* HP7976 tape devices */
MT_7978      0x180002    /* HP7978A & HP7978B tape devices */
MT_7974      0x180003    /* HP7974A tape devices */
MT_7979      0x180004    /* HP7979A tape devices */
MT_7980      0x180005   /* HP7980A & HP7980XC tape devices */
MT_HPIBDDS   0x180006   /* HPIB interface DDS tape devices */
MT_SCSIDDS   0x180007   /* SCSI interface DDS tape devices */
```

Status information will be returned in `mt_gstat`. This is a integer in which the bits represent the following:

bit 00	eof
bit 01	bot
bit 02	eot
bit 03	ssm
bit 04	eod
bit 05	wrt_protect
bit 06	unused
bit 07	online
bit 08	bpi_6250
bit 09	bpi_1600

bit 10	bpi_800
bit 11	unused
bit 12	unused
bit 13	door_open
bit 14	unused
bit 15	immediate_mode
bit 16	bit 31 unused

Implementation Considerations

There will not be any implementation defined items in the magnetic tape portion of `ioctl()`.

There are two operations, `MTNOP`s and `MTEOD`, that will not be supported. The `MTNOP` operation only sets the status, it does not perform an operation. The `MTEOD` operation is used for DDS and QIC devices only, and it does a seek to the "end of data" point. If either of these operations are specified, the `ioctl';' operation will fail and the ENOTTY` error will be set.

Only two of the items in the *mtget* structure will be supported, and their implementation will be as follows. In the Unix implementation of `ioctl`s, the *mt_type* item returns a category or family type of device to the caller. In this implementation of `tioctl`, in addition to returning the device family type, the *mt_type* item will also return the specific tape device type to the caller. The other item that will be supported is *mt_gstat*, and it will return the generic device status as in the Unix implementation. The items that are not supported in the *mtget* structure will be set to 0, and returned to the caller.

If `ioctl()` is interrupted by a signal, the `EINTR` error will be set. Once this function is executing an intrinsic, no signal interruption may occur. Signal interrupts can only occur in the library portion of the code.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The argument *fildes* is not a descriptor for an opened file.
	ACTION	Check to see if *fildes* has been altered or if *fildes* is not initialized.
`EFAULT`	CAUSE	The system detected a NULL address while attempting to use the *arg* parameter passed by the caller.
	ACTION	Check to see if the pointer used is initialized and/or not equal to NULL.
`EINTR`	CAUSE	The `ioctl()` was interrupted by a signal.
	ACTION	Check the state of the file referenced by *fildes*.
`EINVAL`	CAUSE	The *fildes* parameter, the *request* parameter, or the *arg* parameter is invalid. The *fildes* parameter may be zero. The *request* parameter may specify an incorrect operation. The *arg* parameter may specify an unsupported operation for the device.
	ACTION	Validate the parameter values; check if the device is supported.
`EIO`	CAUSE	A physical I/O error occurred on the device.
	ACTION	Check the status of the device.
`ENOTTY`	CAUSE	The specified *request* is not correct for the specified device.
	ACTION	Check to see if the *request* parameter is a correct command. Make sure the *request* is valid for the device.

`ENXIO`	CAUSE	The *request* parameter referenced a device that did not exist, or the *request* made was beyond the limits of the device.
	ACTION	Check to see if the *request* parameter is a correct command. Make sure the *request* is valid for the device.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.
`EROFS`	CAUSE	A write attempt was made to a device that was read-only at the time. This error will be returned for certain devices.
	ACTION	Check the *request* to make sure it is correct for the device specified. Make sure the tape can be written to.
`ESYSERR`	CAUSE	An internal operating system error has occurred; an error not directly applicable to the POSIX functionality.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`ioctl-sockets`, `ioctl-streams`, POSIX.1

ioctl-sockets

Provides an interface and control over magnetic tape devices. In the case of magnetic tape devices, the `ioctl()` function provides an interface for issuing various control commands to opened tape devices. The `ioctl()` operations can be used to position the magnetic tape, and to determine the tape device status.

Syntax

int ioctl(*fildes, request, arg*)
int *fildes*;
int *request*;
void **arg*;

Parameters

fildes The socket descriptor.

request This parameter specifies which command to perform on the socket. The commands are defined in <sys/ioctl.h>. The different commands that are available are described below.

`FIONREAD` Gets the number of bytes that are readable from the socket. For TCP sockets, this is the total number of bytes queued to the socket. For UDP sockets, this is the total number of bytes queued in each datagram and the sum of all the source address structures. The *arg* parameter, will contain the address of the integer with the number of bytes readable.

`FIOSNBIO` Enables or disables non-blocking I/O for the socket. If the integer whose address is *arg* is not zero, then non-blocking I/O is enabled. When non-blocking I/O is enabled, subsequent read and write requests to the socket are prevented from blocking whether the request succeeds or fails. If the integer whose address is *arg* is zero, then non-blocking I/O is disabled.

`FIONBIO` This command is same as the `FIOSNBIO` command.

`FIOGNBIO`	Gets the status of non-blocking i/o. If non-blocking i/o is enabled for the socket, then the integer whose address is *arg* is set to 1. If non-blocking i/o is disabled, the integer is set to zero.
`FIOGSAIO-STAT`	If asynchronous signaling is enabled for the socket, then the integer whose address is *arg* is set to 1. If the asynchronous state is disabled, the integer is set to zero.
`SIOCAT-MARK`	For SOCK_STREAM TCP sockets, upon return if the integer whose address is *arg* is not zero, then the inbound TCP stream has been read up to where the out-of-band data byte starts. If the integer at address *arg* is zero, then the inbound TCP stream has not yet been read up to where the out-of-band data byte starts. For non-TCP sockets, upon return the integer with the address *arg* is always zero.
`SIOCSPGRP`	This command sets the process group or process ID associated with the socket to be the value of the integer whose address is *arg*. If the value of the integer is positive, then a signal is sent to the process with the matching process ID value when the state of the socket changes. If the value is negative, then a signal is sent to all processes that have a process group equal to the absolute value of the specified value when the socket state changes. If the value of the integer with address *arg* is zero, no signal is sent to any processes when the socket state changes.
`SIOCGPGRP`	This command returns the process group or process ID associated with the socket in the integer whose address is *arg*. If the integer is positive, then the value returned corresponds to a process ID. If the integer is negative, then the value returned corresponds to all processes that have a process group equal to the absolute value of that value.

arg This parameter is the address of the integer that the specified *request* needs in order to perform its function. Depending on the type of *request* specified, the integer can represent a variety of values. See the appropriate *request* command for an explanation of the value that the integer will represent in that context.

Return Values

0	The function completes successfully.
-1	If an error occurs, a value of -1 is returned by the function and the global variable `errno` is set with the resultant error.

Description

Sockets are communication endpoints that allow processes to communicate either locally or remotely. For sockets, the `ioctl()` function provides an interface for setting different characteristics for a socket, and retrieving information on a socket.

Implementation Considerations

There will not be any implementation defined items in the sockets portion of `ioctl()`.

There are no mixed environment issues for the sockets portion of `ioctl()`.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The argument *fildes* is not a valid open file descriptor.
	ACTION	Check to see if *fildes* has been altered or if *fildes* is not initialized.

`EFAULT`	CAUSE	The system detected a NULL address while attempting to use the *arg* parameter passed by the caller.
	ACTION	Check to see if the pointer used is initialized and/or not equal to NULL.
`EINTR`	CAUSE	Once this function is executing an intrinsic, no signal interruption may occur. Signal interrupts can only occur in the library portion of the code.
	ACTION	Check the state of the socket referenced by *fildes*.
`EINVAL`	CAUSE	The *request* parameter or the *arg* parameter is invalid, or a socket type that is not supported was specified.
	ACTION	Validate the *request* and *arg* values; check if the socket type is supported.

See Also

`ioctl-streams`, `ioctl-mag_tape()`, POSIX.1

ioctl-streams

Provides an interface and control over magnetic tape devices. In the case of magnetic tape devices, the `ioctl()` function provides an interface for issuing various control commands to opened tape devices. The `ioctl()` operations can be used to position the magnetic tape, and to determine the tape device status.

Syntax

```
int ioctl(fildes, request, arg)
int fildes;
int request;
void *arg;
```

Parameters

fildes The open file descriptor for the stream that will be used.

request This parameter specifies which command to perform on the stream. The commands are defined in <sys/stropts.h>. The different commands that are available are described below.

`FIOGNBIO` Gets the status of non-blocking i/o. If non-blocking i/o is enabled, then the integer whose address is *arg* is set to 1. If non-blocking i/o is disabled, then the integer is set to zero.

`FIONBIO` Enables or disables non-blocking i/o. If the integer whose address is *arg* is not zero, then non-blocking i/o is enabled. When non-blocking i/o is enabled, subsequent read and write requests to the device file are prevented from blocking whether the request succeeds or fails. If the integer whose address is *arg* is zero, then non-blocking i/o is disabled.

`I_ATMARK` Checks to see if the next message is "marked" by the downstream module. If the *earg* value is set to `ANYMARK`, then the check will be to see if

	the message is marked. If the *arg* value is set to `LASTMARK`, then the check will be to see if the message is the last one that is marked on the queue. If marked conditions is satisfied, a 1 is returned; otherwise a zero is returned.
`I_CANPUT`	Checks if a message can be passed on a stream. The *arg* parameter specifies which priority band to check. If the priority band is flow controlled, then a zero is returned; otherwise a 1 is returned.
`I_CKBAND`	Checks if a priority band message is currently on the stream head read queue. The ^fearg^s parameter specifies the priority band being checked. If a message is on the queue, a 1 is returned; otherwise a zero is returned.
`I_FDINSERT`	Creates a message and sends it downstream.
`I_FIFO`	Converts a stream into a FIFO. Used for non-System V systems.
`I_FIND`	Checks for a specific module in the stream. The *arg* parameter will contain the name of the module to be searched for. If the module is present, a 1 is returned; otherwise a zero is returned.
`I_FLUSH`	Flushes the read and/or write queues of the stream depending on the value of the *arg* parameter.
`I_FLUSHBAND`	Flushes a read and/or write band of messages depending on the value of the *arg* parameter. The band of messages to be flushed is also defined in the *arg* structure.
`I_GETBAND`	Gets the priority of the next message on the stream read queue. The priority is returned in the *arg* parameter.

`I_GETCLTIME`	Gets the time delay for closing a stream. The time value is returned in the *arg* parameter.
`I_GETSIG`	Gets the events for which the calling process has registered to receive a signal. The events are returned in the *arg* parameter.
`I_GETSTREAMID`	Gets the stream handle for a C-library file descriptor. Use for NETWARE.
`I_GRDOPT`	Gets the current read mode setting of the stream. The integer value is returned in the *arg* parameter.
`I_GWROPT`	Gets the current write mode setting of the stream. The integer value is returned in the *arg* parameter.
`I_LINK`	Connects two streams. The descriptor of the stream referenced by *fildes* parameter is connected to the descriptor of the stream that is referenced in the *arg* parameter.
`I_LIST`	Gets the list of names of the modules present on the stream.
`I_LOOK`	Gets the name of the first stream module, and places in a character string pointed to by the *arg* parameter.
`I_NREAD`	Returns the number of bytes in the data block of the first message on the stream read queue. The number of bytes is stored in a location pointed to by the *arg* parameter.
`I_PEEK`	Allows the user process to "peek"/look at the first message on the stream read queue. This information will be stored in a location pointed to by the *arg* parameter. If a message is retrieved, a 1 is returned; otherwise a zero is returned.

I_PIPE	Connects two streams as a pipe. Used for non-System V systems.
I_PLINK	Permanently connects two streams. The descriptor of the stream referenced by the *fildes* is connected to the descriptor of the stream referenced by the *arg* parameter. The latter stream is connected via a persistent link that can exist even if the first stream is closed.
I_POP	Removes/pops the module just below the stream head. For this *request*, the *arg* parameter must be set to zero.
I_PUNLINK	Disconnects two streams that are connected via a persistent link.
I_PUSH	Pushes the module whose name is pointed to by the *arg* parameter onto the stream just below the stream head.
I_RECVFD	Retrieves the file descriptor associated with the message sent by the I_SENDFD command over a stream pipe.
I_SENDFD	Requests the stream referred to by *fildes*s to send a message M_PASSFP to the stream head at the other end of a stream pipe.
I_SETCLTIME	Sets the time that the stream head delays when the stream is closing and the write queues contain data. The *arg* parameter contains a pointer to the number of milliseconds to delay.
I_SETSIG	Tells the stream head that the user process wants a SIGPOLL signal to be issued by the kernel for a particular event that can occur on a stream. This command provides support for asynchronous processing in streams. The *arg* parameter contains information that specifies

	the particular events that `SIGPOLL` is to be sent for.
`I_SRDOPT`	Sets the read mode of the stream according to the value of the *arg* parameter.
`I_STR`	Creates an internal`tioctl()` message from the data pointed to by the `arg` parameter and sends the message downstream to a module or driver.
`I-SWROPT`	Sets the stream write mode according to the value of the *arg* parameter.
`I_UNLINK`	Disconnects two streams. One descriptor referenced by the *fildes* parameter, and the other descriptor referenced by the *arg* parameter.

arg This parameter contains additional information that the specified *request* may need to perform its function. This is usually an integer or a pointer to a structure specific to the *request*. See the appropriate *request* command for an explanation of the value that the integer will represent in that context.

Return Values

If the `ioctl` function completes successfully, the following two conditions can occur. If a specific condition is met, then a 1 is returned; if the condition is not met, then a 0 is returned.

If the `ioctl` function does not complete successfully, then a value of -1 is returned by the function and the global variable `errno` is set with the resultant error.

Description

The `ioctl()` commands can be used to perform control operations on streams. User processes can use the commands on all streams file types.

When the stream head receives a streams `ioctl()` function, the *request* and the *arg* parameters are interpreted into an `M_IOCTL` message. In some cases, the parameters are passed onto a module or driver in the stream.

The module in a stream can detect errors in the `ioctl()` function. If an error is detected, an error message containing the error number is sent to the stream head. Subsequent calls to functions will fail with the `errno` set to this number.

Implementation Considerations

None.

Errors

There are many corresponding errors for each of the `request` commands mentioned above. For a list of these errors, and an explanation of the different error conditions, please refer to the HP-UX Release 9.0 manual.

See Also

`ioctl-mag_tape`, `ioctl-streams`, POSIX.1

isatty

Determines whether or not an open file descriptor is associated with a terminal.

Syntax

```
#include <unistd.h>
int isatty (int fildes);
```

Parameters

fildes	An open file descriptor.

Return Values

1	The specified file descriptor is associated with a terminal.
0	The specified file descriptor is not associated with a terminal.
-1	The specified file descriptor is invalid, and `errno` is set to indicate the error condition.

Description

The `isatty()` function returns a value indicating whether or not the open file descriptor *fildes* is associated with a terminal.

Implementation Considerations

Refer to the `EBADF` error description below.

Errors

If an error occurs, `errno` is set to the following value:

`EBADF`	CAUSE	The *fildes* parameter is not a valid open file descriptor.
	ACTION	Check to see if *fildes* has been altered or not initialized.

See Also

POSIX.1 (Section 4.7.2)

kill

Sends a signal to a process or a process group.

Syntax

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig);
```

Parameters

pid A value indicating the process or process group to receive the signal specified in *sig*. Following are valid values and their meanings:

>0	A process whose process ID is equal to *pid*.
0	All processes whose process group ID is equal to the caller's process group ID.
<-1	All processes whose process group ID is equal to the absolute value of *pid*.

If -1 is passed in *pid*, `kill()` fails and sets `errno` to `EINVAL`.

sig A value indicating the signal to be sent. Following are valid values and their meanings:

>0	The signal number of the signal to send. Refer to Table 3-5 for a list of supported signals and their symbolic constants.
0	Test for existence of specified process or process group. (0 is equivalent to the null signal.)

Return Values

0 Success. The `kill()` function had permission to send the signal to at least one of the processes specified in *pid*.

-1 An error occurred. No signal is sent, and `errno` is set to indicate the error condition.

Description

The `kill()` function sends a signal specified by *sig* to a process or group of processes specified by *pid*. If *sig* is 0 (the null signal), no signal is sent, but error checking is performed. Use the null signal to check for the validity of *pid*.

The signal is sent only if the caller has permission to send it to the target process(es). The calling process has permission to send a signal to a target process if one of the following conditions is true:

- The user associated with the calling process has appropriate privileges, defined to be one of the following:
  - A user whose GID matches the GID of the file and who has the MPE/iX account manager (AM) user capability.
  - A user who has the MPE/iX system manager (SM) user capability.
- The signal is `SIGCONT` and the caller's session ID matches the target's session ID.
- The caller's real UID matches either the target's real UID or its saved set-user-ID.
- The caller's effective UID matches either the target's real user ID or its saved set-user-ID.
- The target's UID has been modified by a call to one of the `exec()` functions.

A target process that is blocking a signal does not receive that signal until it unblocks it. (Refer to the `sigaction()` function.) A target process can ignore a signal or install a handler for it. The calling process should not assume that the target process will take the default (or any other) action for the signal.

If the value of *pid* causes *sig* to be generated for the calling process, and if *sig* is not blocked, either *sig* or at least one pending and unblocked signal is delivered to the calling process before the `kill()` function returns.

Implementation Considerations

Job control is not supported.

The {POSIX_SAVED_IDS} constant is always defined.

Use the `kill()` function to send SIGCONT to a process to continue it after SIGSTOP has stopped it. The system never generates SIGCONT and SIGSTOP for a process.

A sending process cannot rely on the target process acting upon a signal in a timely manner if the target process is executing operating system code. The target process is not interrupted until it returns from operating system code.

Refer to Table 3-5 for implementation considerations associated with signals.

Errors

If an error occurs, `errno` is set to one of the following values:

`EINVAL`	CAUSE	The signal *sig* is not a valid signal number, or *pid* is -1.
	ACTION	Refer to Table 3-5 for descriptions of valid signal numbers, or set *pid* to a valid value.
`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or the calling process is in a Procedure Exit handler.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource, or in a Procedure Exit handler.
`ESRCH`	CAUSE	No process or process group matches *pid*.
	ACTION	No action required.

See Also

`getpid()`, `sigaction()`, `<signal.h>`, POSIX.1 (Section 3.3.2).

lseek

Repositions a read/write file offset.

Syntax

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
```

Parameters

fildes	An open file descriptor.
offset	The number of bytes for the new offset. The application of this value is defined by *whence*.
whence	A value specifying how *offset* is to be applied to calculate the resultant offset. Following are valid values and their meanings (defined in `<unistd.h>`).

`SEEK_SET`	Set new offset to *offset*.
`SEEK_CUR`	Set new offset to *offset* plus the current offset.
`SEEK_END`	Set new offset to *offset* plus the current file size.

Return Values

>=0	Success. The new file offset position is returned.
-1	An error occurred. The current offset is not changed, and `errno` is set to indicate the error condition.

Description

The `lseek()` function sets the file offset for the open file description associated with *fildes* to a new position defined by both the *offset* and *whence* parameters. The file offset is the number of bytes from the beginning of the file (where the beginning of the file is file offset 0).

The `lseek()` function allows the file offset to be set beyond the end of existing data in the file. If data is later written at this point, subsequent reads of data in the gap return bytes with the value zero until data is actually written into the gap; however, the `lseek()` function cannot, by itself, extend the size of a file.

Implementation Considerations

Refer to the `ESEEK` and `ESYSERR` error descriptions below.

Pipes (or FIFOs) and device special files are not supported.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *fildes* parameter is not a valid open file descriptor.
	ACTION	Check to see if *fildes* has been altered or is not initialized.
`EINVAL`	CAUSE	The *whence* parameter is not a valid value, or the resulting file offset would be invalid.
	ACTION	Check if value contained by *whence* exceeds the file limit or is a negative value.
`ESEEK`	CAUSE	The *fildes* parameter does not refer to a file that supports seeking.
	ACTION	Certain files or devices do not support seeking. Make sure that the program is not attempting to seek on those files.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`creat()`, `dup()`, `open()`, `read()`, `sigaction()`, `write()`, `<unistd.h>`, POSIX.1 (Section 6.5.3)

mkdir

Creates a directory.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir (const char *pathname, mode_t mode);
```

Parameters

pathname A pointer to a string containing the pathname of the directory to be created. The pathname must be terminated by a null character.

mode The access permission bits for the new directory. Access permission bits are set by ORing any combination of the following macros:

S_IRWXU	Set file owner class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRUSR	Set file owner class read permission bit.
S_IWUSR	Set file owner class write permission bit.
S_IXUSR	Set file owner class execute (if a file) or search (if a directory) permission bit.
S_IRWXG	Set file group class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRGRP	Set file group class read permission bit.
S_IWGRP	Set file group class write permission bit.
S_IXGRP	Set file group class execute (if a file) or search (if a directory) permission bit.
S_IRWXO	Set file other class read, write, and execute (if a file), or search (if a directory) permission bits.
S_IROTH	Set file other class read permission bit.

S_IWOTH	Set file other class write permission bit.
S_IXOTH	Set file other class execute (if a file) or search (if a directory) permission bit.

Unused bits of the *mode* parameter not associated with access permissions must contain zeros or an error occurs.

Return Values

0	Success.
-1	An error occurred. The new directory is not created, and `errno` is set to indicate the error condition.

Description

The `mkdir()` function creates a new directory file whose name is specified in the *pathname* parameter. The newly created directory is an empty directory containing only the directory entries dot (.) and dot-dot (..).

The access permission bits of the new directory are initialized from *mode* and modified by the calling process's file creation mask. The directory's UID is set to the calling process's effective UID. The directory's GID is set to the parent directory's GID.

The `mkdir()` function marks for update the `st_atime`, `st_ctime`, and `st_mtime` time fields of the newly created directory. In addition, `mkdir()` marks for update the `st_ctime` and `st_mtime` time fields of the parent directory.

Implementation Considerations

Refer to the `EFAULT`, `EIMPL`, `ENOSPC`, and `ESYSERR` error descriptions below.

The `S_ISUID` and `S_ISGID` bits are not currently supported.

The `mkdir()` function requires that the calling process have

- write permission to the parent directory
- search permission to each directory component of the pathname
- MPE/iX save files (SF) capability

The `mkdir()` function cannot create the root directory, MPE/iX accounts, or MPE/iX groups.

The `mkdir()` function does not support read-only file systems.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	The calling process does not have search permission to a component of the pathname, or does not have write permission to the parent directory.
	ACTION	Make sure that the calling process has search permission for all components of the pathname and write permission to the parent directory.
`EEXIST`	CAUSE	The directory specified in *pathname* already exists.
	ACTION	Make sure that *pathname* specifies a directory that does not already exist.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.

`EIMPL`	CAUSE	Any of the following conditions: ■ Attempted to create a directory in an MPE/iX account. ■ The directory name exceeded 16 characters in length in the root directory, an MPE/iX account, or an MPE/iX group. ■ The pathname begins with two slash characters (//). ■ Bits of *mode* that are not file permission bits do not contain zeros.
	ACTION	One of the following: ■ Do not create a directory in an MPE/iX account. ■ Make sure that the directory name does not exceed 16 characters in length when in the root directory, an MPE/iX account, or an MPE/iX group. ■ Do not begin a pathname with two slash characters (//). ■ Make sure that bits of *mode* that are not file permission bits contain zeros.
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.

`ENOENT`	CAUSE	A component of the pathname does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOSPC`	CAUSE	The directory could not be created because of a lack of disk space, or the process owner would have exceeded limits imposed by the MPE/iX accounting facility.
	ACTION	Make sure that there is enough space to create the directory on the volume set, or ask your system administrator to increase your accounting limits.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`chmod()`, `stat()`, `umask()`, `<sys/stat.h>`, POSIX.1 (Section 5.4.1)

mkfifo

Make a FIFO special file.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (const char *path, mode_t mode);
```

Parameters

path The pathname of a file.

mode The access permission bits for the new directory. Access permission bits are set by ORing any combination of the following macros:

S_IRWXU	Set file owner class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRUSR	Set file owner class read permission bit.
S_IWUSR	Set file owner class write permission bit.
S_IXUSR	Set file owner class execute (if a file) or search (if a directory) permission bit.
S_IRWXG	Set file group class read, write, and execute (if a file) or search (if a directory) permission bits.
S_IRGRP	Set file group class read permission bit.
S_IWGRP	Set file group class write permission bit.
S_IXGRP	Set file group class execute (if a file) or search (if a directory) permission bit.
S_IRWXO	Set file other class read, write, and execute (if a file), or searc h (if a directory) permission bits.
S_IROTH	Set file other class read permission bit.

`S_IWOTH`	Set file other class write permission bit.
`S_IXOTH`	Set file other class execute (if a file) or search (if a directory) permission bit.

Unused bits of the *mode* parameter not associated with access permissions must contain zeros or an error occurs.

Return Values

0	Successful completion
-1	No FIFO is created, and `errno` is set to indicate the error.

Description

The `mkfifo()` routine creats a new FIFO special file named by the pathname pointed to by *path*. The file permission bits of th enew FIFO are initialized from *mode*. The fil epermission bits of the *moide* argument are modified by the fiel creation mask of the process. When bits in *mode* other than the file permission bits are set, the effect is implementation defined.

The owner ID of the FIFO shall be set to the effective user ID of the process. The group ID of the FIFO shall be set to the group ID of the directory in which the FIFO is being created or to the effective group ID of the process.

Upon successful completion, the `mkfifo()` function shall mark for update the *st_atime*, *st_ctime*, and *st_mtime*% fields of the file. Also, the *st_ctime* and the *st_mtime* fields of the directory taht contains the new entry are ;marked for update.

Implementation Considerations

None.

Errors

If any of the following conditions occur, the `mkfifi9()` function returns -1 and sets `errno` to the corresponding value.

`EACCES`	CAUSE	Serach permission is denied on a compoenent of the path prefix, or wirte permission is denied on the parent directory of the file to be created.
	ACTION	Make sure that the calling process has search permission for all components of the pathname and write permission to the parent directory.
`EEXIST`	CAUSE	The named file already exists.
	ACTION	Make sure that *pathname* specifies a directory that does not already exist.
`ENAMETOOLONG`	CAUSE	The length of the *path* string exceeds [PATH_MAX] , or a pathname component is larger thatn [NAME_MAX] while {_POSIX_NO)TRUNC) is in effect.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	A component of the path prefix does not exists, or the *path* argument points to an empty string.
	ACTION	Specify a valid pathname.
`ENOSPC`	CAUSE	The directory that would contain the new file cannot be extended, or the file system is out of file allocation resources.
	ACTION	Extend accounting limits for the directory in which the file is located, or fr ee up disk space.
`ENOTDIR`	CAUSE	A component of the path prefix is not a directory.
	ACTION	Specify a valid pathname.
`EROFS`	CAUSE	The named file resided on a read-only file system.
	ACTION	Create the *slink* on a writable volume (file system).

See Also

`chmod()`, `exec()`, `pipe()`, `stat()`, <sys/stat.h>, `umask()`, POSIX.1

mknod

Make a FIFO special file.

Makes a directory, or a special or regular file.

Syntax

```
#include <sys/stat.h>
int mknod(const char *path, mode_t mode, dev_t dev);
```

Parameters

path	The pathname of a file.
mode	Specifies the file type and file access permission.
dev	Specifies the major and minor device numbers.

Return Values

0	Successful completion
-1	Error and `errno` is set to indicate the error.

Description

`mknod()` creates a new file named by the path name pointed to by *path*. The mode of the new file is specified by the *mode* argument.

Symbolic constants defining the file type and file access permission bits are found in the <sys/stat.h> header file and are used to construct the *mode* argument. The value of the mode argument should be the bit-wise inclusive OR of the values of the desired file type, miscellaneous mode bits, and access permissions. See `stat(5)` for a description of the components of the file mode.

The owner ID of the file is set to the effective-user-ID of the process. If the set-group-ID bit of the parent directory is set, the new file's group ID is set to the group ID of the parent directory. Otherwise, the new file's group ID is set to the effective-group-ID of the process.

The file access permission bits of mode are modified by the process's file mode creation mask: for each bit set in the process's file mode creation mask, the corresponding bit in the file's mode is cleared (see umask(2)).

The new file is created with three base access-control-list (ACL) entries, corresponding to the file access permission bits.

The dev argument is meaningful only if mode indicates a block or character special file, and is ignored otherwise. It is an implementation- and configuration-dependent specification of a character or block I/O device. The value of dev is created by using the `makedev()` macro defined in <sys/mknod.h>. The `makedev()` macro takes as arguments the major and minor device numbers, and returns a device identification number which is of type `dev_t`. The value and interpretation of the major and minor device numbers are implementation-dependent. For more information, see `mknod(5)` and the System Administration manuals for your system.

Only users with appropriate privileges can invoke `mknod` for file types other than FIFO files.

Implementation Considerations

Proper discretion should be used when using mkrnod to create generic device files in an HP Clustered Environment. A generic device file accessed from different cnodes in a cluster applies to different physical devices. Thus the file's ownership and permissions might not be appropriate in the context of every individual cnode in the cluster.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	One of the following: ■ The directory in which path would be created denies write permission, mode is for a FIFO file and the caller does not have appropriate privileges. ■ A component of the path prefix denies search permission.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of t he pathname. ■ Make sure that the calling process has execute permission to the file.
`EEXIST`	CAUSE	The named *path* already exists.
	ACTION	Make sure that *path* specifies a directory that does not already exist.
`EFAULT`	CAUSE	The path argument points outside the process's allocated address space. The reliable detection of this error is implementation dependent.
	ACTION	Make sure that the pointer is correctly initialized.
`ELOOP`	CAUSE	Too many symbolic links were encountered in translating the path name.
	ACTION	Make sure that there is not a loop in the symbolic links that loops more than `POSIX_SYMLOOP`.
`ENAMETOOLONG`	CAUSE	The length of the specified path name exceeds PATH_MAX bytes, or the length of a component of the path name exceeds NAME_MAX bytes while _POSIX_NO_TRUNC is in effect.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	Either of the following: ■ The path argument is null. ■ A component of the path prefix does not exist.
	ACTION	Specify a valid pathname.

`ENOSPC`	CAUSE	Not enough space on the file system.
	ACTION	Extend accounting limits for the directory in which the file is located, or fr ee up disk space.
`ENOTDIR`	CAUSE	A component of the path prefix is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	The effective-user-ID of the process does not match that of the super-user, and the file type is not FIFO special.
	ACTION	Refer to the `kill()` function description for signal permission rules.
`EROFS`	CAUSE	The directory in which the file is to be created is located on a read-only file system.
	ACTION	Create the *slink* on a writable volume (file system).

See Also

`mknod(1M)`, `chmod(2)`, `exec(2)`, `mkdir(2)`, `setacl(2)`, `umask(2)`, `cdf(4)`, `fs(4)`, `acl(5)`, `mknod(5)`, `stat(5)`, `types(5)`.

open

Opens a file and returns its file descriptor.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (const char *pathname, int oflag, ...);
```

Parameters

pathname A pointer to a string containing a pathname of a file to be opened. The pathname must be terminated by a null character.

oflag A value specifying the file status and file access modes of the file to be opened. If `O_CREAT` is specified, the mode of the file must be passed in a third parameter, *modes*.

The value of *oflag* is the bitwise inclusive OR of flags from the following two lists (defined in `<fcntl.h>`).

Only one of the following three flags must be specified in *oflag*:

`O_RDONLY`	Open for reading only.
`O_WRONLY`	Open for writing only.
`O_RDWR`	Open for both reading and writing.

If the file is opened `O_WRONLY` or `O_RDWR`, the `st_mtime` file time field is marked for update. If the file is opened `O_RDONLY` or `O_RDWR`, the `st_atime` file time field is marked for update.

Any combination of the following optional flags may also be specified in *oflag*:

`O_APPEND` The file offset is set to the end of the file prior to each write.

`O_CREAT` This option requires a third parameter, *mode*, which is of type `mode_t`. If the optional third parameter is not passed when `O_CREAT` is specified, `open()` attempts to read invalid data off the stack, and the results are indeterminate. If the file exists, this flag has no effect, except as noted under `O_EXCL`, below.

If the file is created, the following occurs:

- The file offset is set to the beginning of the file (where the offset position is 0).
- The file's UID is set to the effective UID of the calling process.
- The file's GID is set to the GID of the directory in which the file is being created.
- The access permission bits of the file are set to *mode* and modified by the file mode creation mask of the calling process.
- The following file time fields are marked for update:
  - The file's `st_atime`, `st_ctime` and `st_mtime` time fields.
  - The parent directory's `st_ctime` and `st_mtime` time fields.

`O_EXCL` The file is opened for exclusive access by the calling process. An error results if both `O_EXCL` and `O_CREAT` are specified for an existing file.

An existing file can be opened with `O_EXCL` and without `O_CREAT` only if the file is not currently open by another process (otherwise, an error results).

`O_TRUNC` If the file exists and opened `O_TRUNC` and either `O_RDWR` or `O_WRONLY`, it is truncated to zero length and the mode and owner remain unchanged. The file offset is set to the beginning of the file (where the offset position is 0).

An error results if the file is opened `O_TRUNC` and `O_RDONLY`.

If `O_TRUNC` is specified for an existing file, the `st_ctime` and `st_mtime` time fields of the file are marked for update.

If *oflag* specifies `O_CREAT`, *mode*, a structure of type `mode_t`, must be passed to specify the access permission bits that the file is to be created with. Access permission bits are set by ORing any combination of the following macros:

`S_IRWXU`	Set file owner class read, write, and execute (if a file) or search (if a directory) permission bits.
`S_IRUSR`	Set file owner class read permission bit.
`S_IWUSR`	Set file owner class write permission bit.
`S_IXUSR`	Set file owner class execute (if a file) or search (if a directory) permission bit.
`S_IRWXG`	Set file group class read, write, and execute (if a file) or search (if a directory) permission bits.
`S_IRGRP`	Set file group class read permission bit.
`S_IWGRP`	Set file group class write permission bit.
`S_IXGRP`	Set file group class execute (if a file) or search (if a directory) permission bit.
`S_IRWXO`	Set file other class read, write, and execute (if a file), or search (if a directory) permission bits.
`S_IROTH`	Set file other class read permission bit.
`S_IWOTH`	Set file other class write permission bit.
`S_IXOTH`	Set file other class execute (if a file) or search (if a directory) permission bit.

Bits that are not access permission bits must contain zeros, or an error is returned.

Return Values

>=0	Success. A nonnegative integer is returned representing the lowest numbered file descriptor not open by the calling process.
-1	An error occurred. The file is not opened, and `errno` is set to indicate the error condition.

Description

The `open()` function establishes the connection between a file specified by *pathname* and a file descriptor. It creates an open file description that refers to the file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to the file.

The `open()` function returns a file descriptor for the specified file, which is the lowest file descriptor not currently open for the calling process. The open file description is new; therefore, the file descriptor does not share it with any other process in the system.

Implementation Considerations

Refer to the `EACCES`, `EMFILE`, `EEXCL`, `EFAULT`, `EIMPL`, `EINVAL`, and `ESYSERR` error descriptions below.

Pipes (or FIFOs), device special files, and read-only file systems are not supported through POSIX/iX interfaces and cannot be opened by `open()`. Device files are inherited from the parent process, which has them opened as `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

The GID of a newly created file is set to the GID of the directory in which the file is created.

MPE/iX file equations are ignored by `open()`.

The calling process must have the correct access permissions as defined by either an attached ACD or by the MPE/iX file security matrix. For example, a file opened `O_RDONLY` must have either ACD read access or MPE/iX read access. A file opened `O_WRONLY` or `O_RDWR` must have either ACD write access and append access or MPE/iX write access and append access.

Signals generated for the calling process during execution of `open()` are deferred from delivery until completion of this function.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES` CAUSE One of the following:

- The calling process does not have search permission to a component of the pathname.
- The file does not exist and the calling process does not have write permission to the parent directory of the file to be created.
- The file exists and the permissions specified by *oflag* are denied.
- Both `O_TRUNC` and `O_RDONLY` were specified.
- Both `O_APPEND` and `O_RDONLY` were specified.
- An MPE/iX lockword is associated with the file.

ACTION One of the following:

- Make sure that the calling process has search permission to all directory components of the pathname.
- Make sure that the calling process has write permission to the parent directory of the file to be created.
- Specify valid and compatible flags in *oflag*.
- Remove the MPE/iX lockword.

`EEXCL` CAUSE Attempt to open an existing file exclusively failed because file is already opened.

ACTION Check for ownership of previously opened file. Check file's permission bits.

`EEXIST`	CAUSE	`O_CREAT` and `O_EXCL` are set, and the named file exists.
	ACTION	Open the file a different way, or remove the file.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	One of the following: ■ The specified file is not a byte-stream file. ■ The pathname began with two slash characters (//). ■ Bits in *mode* that are not file permission bits did not contain zeros. ■ An attempt was made to create a file in an MPE/iX account. ■ An attempt was made to create a file with a name that exceeds 16 characters in the root directory or an MPE/iX group.
	ACTION	One of the following: ■ Open only byte-stream files. ■ Do not begin a pathname with two slash characters (//). ■ Set bits in *mode* that are not file permission bits to zero. ■ Do not create files in an MPE/iX account. ■ Do not attempt to create a file with a name which exceeds 16 characters in the root directory or an MPE/iX group.
`EINVAL`	CAUSE	More than one of the following three open flags were specified in *oflag*: `O_WRONLY`, `O_RDONLY`, and `O_RDWR`.
	ACTION	Specify only one of the open flags in *oflag*.
`EISDIR`	CAUSE	The pathname specifies a directory to be opened.
	ACTION	Use `opendir()` to open a directory file.

`EMFILE`	CAUSE	The number of open files and directories would exceed `{OPEN_MAX}`, the limit of opened files per process.
	ACTION	Reduce the number of files and directories opened by the calling process.
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	The `O_CREAT` option is not set, and the named file does not exist; or the `O_CREAT` option is set, and the pathname does not exist; or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOSPC`	CAUSE	Creation of the file would exceed the disk space limits imposed by the MPE/iX accounting facility, or there is not enough free disk space to create the file.
	ACTION	Extend accounting limits for the directory in which the file is located, or free up disk space.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`close()`, `creat()`, `dup()`, `execl()`, `execv()`, `<fcntl.h>`, `lseek()`, `read()`, `<signal.h>`, `fstat()`, `stat()`, `<stat.h>`, `write()`, `umask()`, POSIX.1 (Section 5.3.1)

opendir

Opens a directory stream.

Syntax

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir (const char *pathname);
```

Parameters

pathname A pointer to a string containing a pathname of a directory to open. The pathname must be terminated by a null character.

Return Values

<>NULL	Success. A pointer to an object of type `DIR` is returned.
NULL	An error occurred. The directory is not opened, and `errno` is set to indicate the error condition.

Description

The `opendir()` function opens a directory stream associated with the directory specified by *pathname* and returns a pointer to the open directory stream to be used by subsequent calls to `readdir()`, `rewinddir()`, and `closedir()`.

The directory stream is positioned at the first entry in the directory.

Implementation Considerations

Refer to the `EFAULT`, `EIMPL`, `EMFILE`, and `ESYSERR` error descriptions below.

The type `DIR` (defined in `<dirent.h>`) is implemented using a file descriptor. Applications can only open a total of `{OPEN_MAX}` files and directories.

The `FD_CLOEXEC` flag is not currently supported.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	Either the calling process does not have search permission to a component of *pathname* or does not have read permission to the directory to be opened.
	ACTION	Make sure that the calling process has ACD traverse directory (TD) access for all components of the pathname and ACD read directory (RD) access to the directory to be opened.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	The pathname begins with two slash characters (//).
	ACTION	Do not begin pathnames with two slash characters (//).
`EMFILE`	CAUSE	The number of directory streams and files opened by the calling process would exceed `{OPEN_MAX}`.
	ACTION	Reduce the number of directories and files opened by the process.
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	The specified directory does not exist, or *pathname* points to an empty string.
	ACTION	Specify an existing directory name.

`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`closedir()`, `readdir()`, `rewinddir()`, `<dirent.h>`, POSIX.1 (Section 5.1.2)

pause

Suspends execution of the calling process.

Syntax

```
#include <unistd.h>
int pause (void);
```

Parameters

None.

Return Values

No return	Success.
-1	An error occurred, and `errno` is set to indicate the error condition.

Description

The `pause()` function suspends execution of the calling process until the delivery of a signal that either executes a user-supplied signal handling function (signal handler) or causes the process to terminate. If the signal executes a signal handler, `pause()` returns a -1 after the signal handler returns. If a signal terminates the paused process, `pause()` does not return to the caller.

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to the following value:

`EINTR`	CAUSE	A signal was caught by the calling process, and control was returned from the signal handler.
	ACTION	No action required.

See Also

`alarm()`, `kill()`, `sigaction()`, `wait()`, POSIX.1 (Section 3.4.2).

pathconf

Gets configuration variable for path name.

Syntax

```
#include <unistd.h>
long pathconf(char *pathname, int name);
```

Parameters

pathname	is the name of the file or directory.
name	is a symbol indicating the variable, the value of which you want to determine, relative to the file or directory specified in pathname. .

Description

`pathconf()` lets you determine the value of a configuration variable associated with a particular file. If `pathconf()` can determine the value of the requested variable, it returns that value as its result.

The name argument may be any one of a set of symbols defined in <unistd.h>. Each symbol stands for a configuration variable. The following list shows the possible symbols and the variables that each symbol stands for:

`_PC_LINK_MAX`	stands for `LINK_MAX` defined in <limits.h>—the maximum number of links the file can have. If *pathname* is a directory, `pathconf()` returns the maximum number of links which can be established to the directory itself.
`_PC_MAX_CANON`	stands for `MAX_CANON` defined in <limits.h>—the maximum number of bytes in a terminal canonical input line. *pathname* must refer to a terminal file.
`_PC_MAX_INPUT`	stands for `MAX_INPUT` defined in <limits.h>—the minimum number of bytes for which space is available in a terminal input queue, which means the maximum number of bytes that a portable

application may have the user enter before the application actually reads the input. *pathname* must refer to a terminal file.

`_PC_NAME_MAX`
: stands for `NAME_MAX` defined in <limits.h>—the maximum number of characters in a filename (not including any terminating `0` if the filename is stored as a string). This only refers to the filename itself, that is, the last component of the file's path name. *pathname* must be a directory, and `pathconf()` returns the maximum length of filenames for files in the directory.

`_PC_PATH_MAX`
: stands for `PATH_MAX` defined in <limits.h>—the maximum number of characters in a complete path name (not including any terminating \0 if the path name is stored as a string). *pathname* must be a directory, and `pathconf()` returns the maximum length of a relative path name when the specified directory is the working directory.

`_PC_PIPE_BUF`
: stands for `PIPE_BUF` defined in <limits.h>—the maximum number of bytes that can be written 'atomically' to a pipe. If more than this number of bytes are written to a pipe, the operation may take more than one physical write operation and may require more than one physical read operation to read the data on the other end of the pipe. If *pathname* is a FIFO file, `pathconf()` returns the value for the file itself. If *pathname* is a directory, `pathconf()` returns the value for any FIFOs which exist or can be created under the directory. If *pathname* is any other kind of file, the behavior is undefined.

`_PC_CHOWN _RESTRICTED`
: stands for `_POSIX_CHOWN_RESTRICTED` defined in the <unistd.h>. This indicates that the use of the `chown()` function is restricted—see `chown()` for more details. If *pathname* is a directory, `pathconf()` returns the value for any kind of file

	under the directory, but not for subdirectories of the directory.
`_PC_NO_TRUNC`	stands for `_POSIX_NO_TRUNC` defined in <unistd.h>. This indicates that an error is to be generated if a file name is longer than `NAME_MAX`. *pathname* must be a directory, and the value returned by `pathconf()` applies to all files under that directory.
`_PC_VDISABLE`	stands for `_POSIX_VDISABLE` defined in <unistd.h>. This indicates that terminal special characters can be disabled using this character value, if it is defined; see `tcsetattr()` for details. *pathname* must refer to a terminal file.

For `_POSIX_CHOWN_RESTRICTED`, `_POSIX_NO_TRUNC`, and `_POSIX_VDISABLE`, `pathconf()` returns -1 if the option is turned off and another value otherwise.

If a particular variable has no limit (for example `PATH_MAX`), `pathconf()` returns -1 but does not change `errno`.

Errors

If `pathconf()` cannot determine an appropriate value, it returns -1 and sets errno to one of the following:

`EACCES`	CAUSE	The process does not have search permission on some component of the *pathname* prefix.
	ACTION	Ensure that the process has search permissions on all components of the *pathname* prefix.
`EINVAL`	CAUSE	*name* is not a valid variable code, or the given variable cannot be associated with the specified file.
	ACTION	Ensure that *name* is a valid variable code.

`ENAMETOOLONG`	CAUSE	*pathname* is longer than `PATH_MAX` characters, or some component of *pathname* is longer than `NAME_MAX` and `_POSIX_NO_TRUNC` is set.
	ACTION	Unset the configuration variable `_POSIX_NO_TRUNC' to disable checking the length of` *pathname* `or modify` *pathname* `to ensure that the entire name is less than PATH_MAX` characters in length and that each component is less than `NAME_MAX` characters in length.
`ENOENT`	CAUSE	There is no filenamed *pathname*, or the *pathname* argument is an empty string.
	ACTION	Ensure that you provide a *pathname* and that *pathname* is a valid file.
`ENOTDIR`	CAUSE	Some component of the *pathname* prefix is not a directory.
	ACTION	Ensure that all components of *pathname* are valid directories.

See Also

`fpathconf()`

pclose

Close a pipe.

Syntax

```
#include <stdio.h>
int pclose(FILE *stream);
```

Parameters

stream	is the pointer for a pipe opened with `popen()`. If it is not a pointer for a pipe opened with `popen()`, the result is undefined.

Description

`pclose()` closes a pipe that was opened with `popen()`. It then waits for the command on the other end of the pipe to terminate.

Errors

Normally, `pclose()` returns the termination status of the command at the other end of the pipe. However, if the process calling `pclose()` has also called `wait()` or `waitpid()` with a pid argument less than or equal to zero, or with some non-standard function that makes it impossible for `pclose()` to determine the termination status, `pclose()` returns -1 and sets errno to ECHILD.

If `popen()` was unable to invoke the shell to execute a command, `pclose()` returns a termination status as if the shell had terminated with `exit(127)`.

`pclose()` may set `errno` to one of the following:

`ECHILD`	CAUSE	`pclose()` was unable to determine the child process's status.
	ACTION	No action is necessary.

See Also

sh(1), popen()

pipe

Create an inter-process channel.

Syntax

```
int pipe (int fildes[2])
```

Parameters

fildes An open file descriptor.

Return Values

0 successful completion

-1 error, `errno` is set to indicate the error

Description

The `pipe()` function creates a pipe and places two file descriptors, one each into the arguments *fildes*[0] and *fildes*[1]. These arguments refer to the open file descriptions for the read and write ends of the pipe. Their integer values are the two lowest available at the time of the `pipe()` function call. The O_NONBLOCK and FD_CLOEXEC flags are clear on both file descriptors. The `fcntl()` function can be used to set these flags.

Data is written to file descriptor *fildes*[1] and read from file descriptor *fildes*[0]. A read on file descriptor *fildes*[0] accesses the data written to file descriptor *fildes*[1] on a first-in-first-out basis.

A process has the pipe open for reading if the read end file descriptor, *fildes*[0], is open. A process has the pipe open for writing if the write end file descriptor, *fildes*[1] is open.

Upon successful completion, the `pipe()` function marks for update the `st_atime`, `st_ctime`, and `st_mtime` fields of the pipe.

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to one of the following values:

`EMFILE`	CAUSE	More than {OPEN_MAX}-2 file descriptors are already in use by this process.
	ACTION	Check process limit in `<limits.h>`. Close a file.
`ENFILE`	CAUSE	The number of simultaneously open files in the system would exceed a system-imposed limit.
	ACTION	Close a file.

See Also

`fcntl()`, `open()`, `read()`, `write()`, POSIX.1

popen

Open a pipe to a command and execute the command.

Syntax

```
#include <stdio.h>
FILE *popen(const char *command, const char *mode);
```

Parameters

command	Is a string giving the command line for a command you want to execute.
mode	Specifies the nature of the pipe you want to open. This can be the string "r" or "w". See the following section for more details.

Description

`popen()` executes the command specified by command. It does this as if it spawns a child process with `fork()`, then the child process invokes the shell sh with

```
execl (shellpath, "sh", "-c", command, NULL);
```

where *shellpath* is the path name of the file that contains the shell.

`popen()` establishes a pipe between *command* and the process which executes `popen()`. The result of `popen()` is a FILE * pointer that can be used to read/write on this pipe. If *mode* is "r", standard output from *command* is piped to the process which calls `popen()`. Data shipped along this pipe can be read with normal I/O calls using the FILE * pointer returned by `popen()`. If *mode* is "w", output written to the pipe by the process which calls `popen()` is sent as the standard input to *command*.

Streams opened with `popen()` should be closed with `pclose()`.

Errors

`popen()` returns `NULL` if it cannot create the pipe or the child process. It sets `errno` to one of the values used by `pipe()` or `fork()`. `popen()` may also set `errno` to:

`EINVAL`	CAUSE	The value of mode was invalid.
	ACTION	Specify a valid value for mode.

See Also

`sh(1)`, `pclose()`

read

Reads data from a file.

Syntax

```
#include <unistd.h>
ssize_t read (int fildes, void *buffer, size_t nbyte);
```

Parameters

fildes	An open file descriptor.
buffer	A pointer to a buffer where data is returned. The size of the buffer must be greater than *nbyte*.
nbyte	The maximum number of bytes to read.

Return Values

>=0	Success. An integer value indicating the number of bytes actually read is returned.
-1	An error occurred. The content of the buffer is indeterminate and `errno` is set to indicate the error condition.

Description

The `read()` function attempts to read *nbyte* bytes from the open file associated with the open file descriptor *fildes* into the buffer pointed to by *buffer*.

On a file capable of seeking, `read()` starts from the current file offset position. Before successful return, the file offset is incremented by the number of bytes actually read.

On a file not capable of seeking, `read()` starts from the current position. (The file offset for such a file is undefined.)

Upon successful completion, the `read()` function returns the actual number of bytes copied to the buffer and, if *nbyte* is greater than 0, marks for update the `st_atime` time field of the file.

The value returned by `read()` is never greater than *nbyte*. The value returned may be less than *nbyte* if either the number of bytes left in the file is less than *nbyte* or the file is a special file (`STDIN_FILENO`) and fewer than *nbytes* are available.

If *nbytes* is zero, the `read()` function returns zero bytes of data. In this case, the file offset position is not changed, and no time fields are marked for update.

No data transfer occurs past the current end-of-file (EOF). Zero is returned if the file offset position is at or after the EOF. For any portion of the file prior to the EOF that is not written to, `read()` returns bytes with a value of zero.

Implementation Considerations

Refer to the `EFAULT`, `EIMPL`, and `ESYSERR` error descriptions below.

Signals generated for the calling process during execution of `read()` are deferred from delivery until completion of this function.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *fildes* parameter is not a valid file descriptor open for reading.
	ACTION	Check the value of the *fildes*, check permission bits of file or check the access mode used in opening the file.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *buffer* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	The value specified in *nbyte* is greater than `{SSIZE_MAX}`.
	ACTION	Reduce the value specified in *nbyte*.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`creat()`, `dup()`, `execl()`, `execv()`, `fork()`, `open()`, `unlink()`, POSIX.1 (Section 6.4.1)

readdir

Reads entries from an open directory stream.

Syntax

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir (DIR *dirp);
```

Parameters

dirp	A pointer to an open directory stream obtained from a successful call to `opendir()`.

Return Values

<>NULL	Success.
NULL	End of directory stream was reached, but `errno` is not modified.
NULL	An error occurred, and `errno` is set to indicate the error condition.

Description

The `readdir()` function returns a pointer to a structure of type `dirent` representing the directory entry at the current position in the directory stream associated with *dirp*, then positions the directory stream at the next entry. A NULL pointer is returned upon reaching the end of the directory stream.

Upon successful completion, `readdir()` marks for update the `st_atime` time field of the directory.

The pointer returned by `readdir()` points to data that is overwritten by another call to `readdir()` on the same directory stream.

Implementation Considerations

Refer to the `EFAULT` and `ESYSERR` error descriptions below.

Both the dot and dot dot directory entries are returned only for directories that explicitly contain them. The root directory, MPE/iX accounts, and MPE/iX groups do not contain explicit dot and dot dot entries.

If an entry is removed from or added to the directory after the most recent call to `opendir()` or `rewinddir()`, subsequent returns from `readdir()` accurately reflect the current state of the directory.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *dirp* parameter does not refer to an open directory stream.
	ACTION	Pass an open directory stream pointer returned by the `opendir()` function.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *dirp* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`mkdir()`, `closedir()`, `opendir()`, `rewinddir()`, `<dirent.h>`, POSIX.1 (Section 5.1.2)

readlink

Reads the value of a symbolic link.

Syntax

```
#include <unistd.h>
init readlink(const char *path, char *buf, size_t bufsiz);
```

Parameters

path The pathname of a file.

buf Points to the region of memory where `confstr()` stores the string value of the variable indicated by name.

len Is the maximum number of characters that can be placed in buf. If this is not enough to hold the complete string value of name, `confstr()` truncates the string value to len-1 characters and appends a null terminator (the \0 character).

Return Values

Upon successful completion, the *readlink()* function will return the number of bytes placed in the buffer when *bufsiz* is greater than zero, or the number of bytes contained in the symbolic link when *bufsiz* is equal to zero. If the return value is equal to *bufsiz*, the buffer need not contain the entire contents of the symbolic link; for *bufsiz* can be used to determine the size of the contents of the symbolic link. If the *readlink()* function is unsuccessful, a value of -1 will be returned and `errno` will be set to indicate the error.

Description

The *readlink* function will place the contents of the symbolic link, *path*, in the buffer *buf*, which has size *bufsiz*. The contents of the returned symbolic link will not include a null terminator. As a special case, if the value of *bufsiz* is 0, no change will occur to the buffer *buf* and *readlink()* will return the number of bytes contained in the symbolic link.

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file.
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.

`ELOOP`	CAUSE	A loop exists in symbolic links encountered during resolution of the *path* argument. This error may be returned if more than {POSIX_SYMLOOP} symbolic links are encountered during resolution of the *path* argument.
	ACTION	Make sure that there is not a loop in the symbolic links that loops more than `POSIX_SYMLOOP`.
`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, ot *pathname* points to an empty string.
	ACTION	Specify a valid pathname.

See Also

`stat()`, `lstat()`, `symlink()`

regcomp

Compile a regular expression.

Syntax

```
#include <regex.h>
int regcomp (regex_t *reg, const char *regstr, int flags);
```

Parameters

reg — points to an object where `regcomp()` stores the compiled regular expression. `regex_t` is defined in <regex.h>.

regstr — points to the regular expression as a string (the way it might be specified for a command like grep).

flags — gives a variety of flags for the compilation. Flags are given by symbols defined in <regex.h> which can be ORed together. The recognized symbols are:

- `REG_EXTENDED` — uses extended regular expressions (see `regexp(3)`). The default is to interpret *regstr* as a basic regular expression.
- `REG_ICASE` — ignores the case of letters in matches. The setting of `LC_CTYPE` affects which characters are considered to be opposite cases of each other.
- `REG_NEWLINE` — treats the newline character as a regular character, without its special meaning.
- `REG_NOSUB` — indicates that `regcomp()` should only report success or failure, and not set *reg*`->re_nsub` (see the following section). It

also affects the behavior of `regexec(3)`

Return Values

0	Successful compile
Error code	Not successful compile

Description

`regcomp()` compiles a regular expression for later use. Early implementations of `regcomp()` generated executable code that determined whether or not strings matched *regstr*. Under POSIX.2, `regcomp()` may generate executable code and/or data which speeds pattern-matching. The result of `regcomp()` is a structure of the `regex_t type` which is stored in *reg*. This structure type contains at least the following field:

`size_t re_nsub` is usually set to the number of parenthesized subexpressions found in *regstr*. These subexpressions are delimited with

```
\(\)
```

in basic regular expressions and

```
()
```

in extended regular expressions. `regcomp()` does not set `re_nsub` if `REG_NOSUB` is turned on in *flags*.

Errors

If `regcomp()` successfully compiles *regstr* it returns zero; otherwise, it returns one of the following symbolic values:

`REG_BADBR`	CAUSE	The contents of `\{\}` were invalid: not a number, too large a number, more than two numbers, first number larger than second.
	ACTION	Make sure that the contents of `\{\}` or `{}` are valid.

`REG_BADPAT`	CAUSE	*regstr* was an invalid regular expression.
	ACTION	Specify a valid regular expression
`REG_BADRPT`	CAUSE	*regstr* contained a ?, *, or + that was not preceded by a valid regular expression.
	ACTION	Make sure that every unquoted /, *, or + in regstr is preceded by a valid regular expression.
`REG_EBRACE`	CAUSE	regstr contained a `\{\}` imbalance.
	ACTION	Make sure that all `{` and `}` characters and all `\{` and `\}` characters appear in matched pairs in *regstr*.
`REG_EBRACK`	CAUSE	regstr contained a `[]` imbalance.
	ACTION	Make sure that all `[` and `]` characters appear in matched pairs in *regstr*.
`REG_ECOLLATE`	CAUSE	*regstr* contained a reference to an invalid collating element.
	ACTION	Make sure that all collating elements referenced in regstr are valid in the locale indicated by `LC_COLLATE`.
`REG_ECTYPE`	CAUSE	regstr contained a reference to an invalid character class.
	ACTION	Make sure that all character classes referenced in regstr are valid in the locale indicated by `LC_CTYPE`.
`REG_EESCAPE`	CAUSE	regstr contained a trailing \.
	ACTION	Remove the trailing \ or complete the escape sequence.
`REG_ENEWLINE`	CAUSE	A newline was found before the end of a pattern, and the `REG_ENEWLINE` flag was not set.
	ACTION	Set the `REG_ENEWLINE` flag, or check the pattern for a missing `/`.
`REG_EPAREN`	CAUSE	regstr contained a `()` or `\(\)` imbalance.
	ACTION	Make sure that all `(` and `)` characters and all `\(` and `\)` characters appear in matched pairs in regstr.

`REG_ERANGE`	CAUSE	A range expression contained an invalid endpoint. For example, an equivalence or character class is not valid.
	ACTION	Specify a valid endpoint.
`REG_ESPACE`	CAUSE	There were not enough free system resources for **`regcomp()`** to compile *regstr*
	ACTION	Free up more resources or specify a less complex regular expression.
`REG_ESUBREG`	CAUSE	The *number* in a *number* construct was greater than the number of matching subexpressions.
	ACTION	Make sure that *number* is less than or equal to the number of matching subexpressions.
`REG_EFATAL`	CAUSE	An internal error occurred.
	ACTION	Contact your system manager.

See Also

`grep(1)`, `regexec()`, `regfree()`, `regexp(3)`

regerror

Convert regular expression errors to messages.

Syntax

```
#include <sys/types.h>
#include <regex.h>
size_t regerror(int errcode, const regex_t *reg,
 char *errbuf, size_t len)
```

Parameters

errcode Is the last non-zero value returned by a call to `regcomp()` or 'regexec()".

reg Points to an object where `regcomp()` stored a compiled regular expression. `regex_t` is defined in <regex.h>. *reg* is currently unused in this implementation.

errbuf Points to the region of memory where `regerror()` stores the generated error message.

len Is the maximum number of characters that can be placed in errbuf. If this is not enough to hold the generated message, `regerror()` truncates the message to len-1 characters and appends a `\0` character.

Description

`regerror()` takes an error code produced by `regcomp()` or `regexec()` and produces a printable error message that corresponds to the error condition. The return value of `regerror()` is the length of this error message.

If the len argument is not zero, `regerror()` places the error message in the buffer pointed to by errbuf, truncating it, if necessary. If the len is zero, `regerror()` ignores the buffer, but still returns the length of the appropriate error message.

Erorrs

`regerror()` normally places one of the messages from the `regerror(3)` man page in the buffer indicated by *errbuf. All messages are shown with the error code returned by `regcomp()` or regexec()".

See Also

`regexec()`, `regfree()`, `regexp(3)`

regexec

Compare string against compiled regular expression.

Sytnax

```
#include <sys/types.h>
#include <regex.h>
int regexec(const regex_t *reg, const char *string,
 size_t maxmatch, regmatch_t submatch[] ,
 int flags);
```

Parameters

reg Must point to an object where `regcomp()` stored a compiled regular expression.

string Is the string you want to compare against the regular expression associated with *reg*.

maxmatch Is the maximum number of matching substrings that you want `regexec()` to find. This should be less than or equal to the number of elements that can be stored in the *submatch* array.

submatch[] Points to an array with a length of at least *maxmatch* where `regexec()`strings of *string* which match the regular expression *reg*.

flags Holds flags that affect the behavior of `regexec()`. Flag values are represented by symbolic constants defined in <regex.h>. To get an appropriate *flags* value, OR the desired symbols together. Possible symbols are:

`REG_NOTBOL` Tells `regexec()` not to treat the beginning of *string* as the beginning of the text line. This means that the special meaning of the caret (^) (the beginning of the line) never matches in *string*.

`REG_NOTEO` Tell `regexec()` not to treat the end of *string* as the end of the text line. This means that the special meaning of the dollar sign ($) (the end of the line) never matches in *string*.

Return Values

`regexec()` returns zero to indicate a successful match, or one of the following error codes.

Description

`regexec()` compares the given string to the regular expression *reg*. *reg* must have been created by a previous call to `regcomp()`. The `regcomp()` flags that were specified at the time the regular expression was compiled affect the results of as specified in the flags to `regcomp()` or maxmatch is zero, `regexec()` simply checks whether or not the given *string* contains a match for *reg*. If so, `regexec()` returns zero. If there is no match, `regexec()` returns the value of `REG_NOMATCH`, defined in <regex.h>. When `REG_NOSUB` is in effect, *maxmatch* should be zero.

If `REG_NOSUB` was not specified, `regexec()` uses the array *submatch* to record substrings matching the regular expression. The elements of this array have the structure type `regmatch_t`, defined in <regex.h>. This structure contains at least the following:

`char *rm_sp;`	points to the first character of a matching substring.
`char *rm_ep;`	points to the character immediately following the end of a matching substring.
`off_t rm_so;`	offset from string to the first character of a matching substring.
`off_t rm_eo;`	offset from string to the character immediately following the end of a matching substring.

submatch[0] contains the first substring of *string* that matches the entire regular expression *reg*. If *reg* contains parenthesized subexpressions, *submatch[i]* contains the substring matching the ith parenthesized subexpression. For example, if you have a Basic regular expression

```
a\(b*\)c\(d*\)
```

submatch[0] contains the match for the whole regular expression, *submatch[1]* contains the match for `\(b*\)`, and *submatch[2]* contains the match for `\(d*\)`. Unused elements of *submatch* have NULL pointers and -1 offsets. If there are more than *maxmatch* matching substrings, `regexec()` finds them but only records *maxmatch* in *submatch*.

A parenthesized subexpression of *pattern* might be part of several different substring matches, or match nothing even though the expression as a whole matches. In this case, `regexec()` follows these rules:

1. If subexpression i participated in the match several times, *submatch[i]* refers to the last such match.
2. If subexpression *i* did not participate in a successful match, the pointers in *submatch[i]* are NULL and the byte offsets in *submatch[i]* are set to -1. This can happen, for example, if the regular expression has the form `A|B`; if *string* matches the `A` part and subexpression *i* appears in the `B` part, there is no match for subexpression *i*.
3. If subexpression *i* is contained within another subexpression *j*, no other subexpression within *j* contains *i*, and *submatch[j]* reports a match of subexpression *j*, then *submatch[i]* reports the match or nonmatch of subexpression *i* as described in rules 1 and 2, but within the substring reported in *submatch[j]* rather than the whole string.
4. If subexpression *i* is contained in subexpression *j*, and the pointers in *submatch[j]* are NULL, the pointers in *submatch[i]* are also NULL and the byte offsets in *submatch[i]* are set to -1.
5. If subexpression *i* matches a zero-length string, both pointers and both byte offsets in *submatch[i]* indicate the character after the zero-length string.

Errors

If an error occurs, `errno` is set to one of the following values:

`REG_ESPACE`	CAUSE	There were not enough free system for `regexec()` to carry out the comparison.
	ACTION	Free up more resources, or specify a less complex regular expression or shorter string.
`REG_NOMATCH`	CAUSE	No match was found.
	ACTION	No action is required.
`REG_EFATAL`	CAUSE	Some other error occurred. For example, *reg* was not a valid compiled regular expression, or was destroyed by an errant pointer.
	ACTION	Check your program carefully.

See Also

`grep(1)`, `regcomp()`, `regfree()`, `regexp(3)`

regfree

Free a compiled regular expression.

Syntax

```
#include <regex.h>
void regfree(regex_t *reg);
```

Parameters

reg must point to an object where `regcomp()` stored a compiled regular expression.

Description

`regfree()` frees any memory that was allocated by `regcomp()` when it compiled the regular expression associated with *reg*.

Note The `regex_t` object itself is not freed.

See Also

`regcomp()`, `regexec()`, `regexp(3)`

rename

Renames an existing file.

Syntax

```
#include <stdio.h>
int rename(const char *old, const char new);
```

Parameters

old The pathname of the file to be renamed.

new The new pathname of the file.

Return Values

Upon successful completion, a value of zero will be returned, Otherwise, a value of -1 will be returned and `errno` will be set to indicate the error. If -1 is returned, neither the filenamed by *old* nor the filenamed by *new*, if either exists, will be changed by this function call.

Description

The *rename()* function changes the name of a file. The *old* argument points to the pathname of the file to be renamed, The *new* argument points to the new pathname of the file.

If the *old* argument and the *new* argument both refer to links to the same existing file, The *rename()* function will return successfully and perform no other action.

The *old* and *new* arguments must be of the same type of file or directory. If the link named by the *new* argument exists, it will be removed and *old* renamed to *new*. Write access permission is required for both the directory containing *old* and the directory containing *new*.

If the *old* argument points to the pathname of a directory, the *new* argument will not point to the pathname of a file that is not a directory. If the directory named by the *new* argument exists, it will be removed and *old* renamed to *new*. Thus, if *new* names an existing directory, it will be required to be an empty directory.

The *new* pathname should not contain a path prefix that names *old*.

If the link named by the *new* argument exists and the link count of the file becomes zero when it is removed and no process has the file open, the space occupied by the file will be freed and the file will no longer be accesable. If one or more processes have the file open when the last link is removed, the link will be removed before *rename()* returns, but the removal of the file contents will be postponed until all references to the file have been closed.

Upon successful completion, the *rename()* function will mark for update the *st_ctime* and *st_mtime* fields of the parent directory of each file.

Errors

If an error occurs, `errno` is set to one of the following values:

`EINVAL`	CAUSE	More than one of the following three open flags were specified in *oflag*: `O_WRONLY`, `O_RDONLY`, and `O_RDWR`.
	ACTION	Specify only one of the open flags in *oflag*.
`EISDIR`	CAUSE	The pathname specifies a directory to be opened.
	ACTION	Use `opendir()` to open a directory file.
`EMFILE`	CAUSE	The number of open files and directories would exceed `{OPEN_MAX}`, the limit of opened files per process.
	ACTION	Reduce the number of files and directories opened by the calling process.

`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOENT`	CAUSE	The `O_CREAT` option is not set, and the named file does not exist; or the `O_CREAT` option is set, and the pathname does not exist; or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOSPC`	CAUSE	Creation of the file would exceed the disk space limits imposed by the MPE/iX accounting facility, or there is not enough free disk space to create the file.
	ACTION	Extend accounting limits for the directory in which the file is located, or free up disk space.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`close()`, `creat()`, `dup()`, `execl()`, `execv()`, `<fcntl.h>`, `lseek()`, `read()`, `<signal.h>`, `fstat()`, `stat()`, `<stat.h>`, `write()`, `umask()`, POSIX.1 (Section 5.3.1)

rewinddir

Resets an open directory stream to point to the first entry of the directory.

Syntax

```
#include <sys/types.h>
#include <dirent.h>
int rewinddir (DIR *dirp);
```

Parameters

dirp	A pointer to an open directory stream obtained from a successful call to `opendir()`.

Return Values

0	The position is successfully reset.
-1	An error occurred. The current position is not changed, and `errno` is set to indicate the error condition.

Description

The `rewinddir()` function resets the position of the directory stream to which *dirp* refers to the first entry of the directory. It also causes the directory stream to refer to the current state of the directory, as a call to `opendir()` does.

Implementation Considerations

Refer to the `EFAULT` error description below.

The return type of `rewinddir()` is `int` to be able to return a value indicating an error. The POSIX.1 standard calls for no value to be returned (`void`). A strictly conforming POSIX application should not evaluate the return of `rewinddir()`.

The type `DIR` is implemented using a file descriptor.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *dirp* parameter does not refer to an open directory stream.
	ACTION	Pass an open directory stream pointer returned by the `opendir()` function.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *dirp* parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`mkdir()`, `closedir()`, `opendir()`, `readdir()`, `<dirent.h>`, POSIX.1 Section 5.1.2.

rmdir

Purges (removes) a directory.

Syntax

```
#include <unistd.h>
int rmdir (const char *pathname);
```

Parameters

pathname A pointer to a string containing a pathname of the directory to purge. The pathname must be terminated by a null character.

Return Values

0 Success.

-1 An error occurred. The directory is not removed, and `errno` is set to indicate the error condition.

Description

The `rmdir()` function purges (removes) the directory specified by *pathname*. The directory is removed only if it is an empty directory (containing only the dot and dot dot directory entries).

If the link count of the directory becomes zero, and no process has the directory open, the directory is purged from the system and is no longer accessible.

If one or more processes have the directory open when the last link is removed, the dot and dot dot entries are removed before `rmdir()` returns and no new entries can be created; however, the directory is not purged until all references to the directory have been closed.

Upon successful completion, `rmdir()` marks for update the `st_ctime` and `st_mtime` time fields of the parent directory.

Implementation Considerations

Refer to the `EFAULT`, `EIMPL`, and `ESYSERR` error descriptions below.

The `rmdir()` function cannot remove the dot and dot dot directory entries, the root directory, MPE/iX accounts, or MPE/iX groups.

Errors

If an error occurs, `errno` is set to one of the following values:

`EACCES`	CAUSE	The calling process either does not have search permission to a component of the pathname or does not have write permission to the parent directory.
	ACTION	Make sure that the calling process has search permission to all components of the pathname and write permission to the parent directory.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
`EIMPL`	CAUSE	An attempt was made to remove the dot or dot dot directory entries, the root directory, an MPE/iX account, or an MPE/iX group; or the pathname began with two slash characters (//).
	ACTION	Remove MPE/iX accounts and MPE/iX groups using MPE/iX CI commands. The root directory can never be removed. Do not begin pathnames with two slash characters (//).
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.

`ENOENT`	CAUSE	The specified directory does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ENOTEMPTY`	CAUSE	The directory specified by *pathname* cannot be removed because it is not empty.
	ACTION	Make sure that the directory is an empty directory.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`mkdir()`, `unlink()`, POSIX.1 (Section 5.5.2)

setuid

Sets user IDs.

Syntax

```
#include <sys/types.h>
init setuid(uid_t uid);
```

Parameters

uid The ID of a user.

Return Values

Upon successful completion, a value of zero is returned. If unsuccessful, a value of -1 is returned and `errno` is set to indicate the error.

Description

If {_POSIX_SAVED_IDS} is defined:

If the process has appropriate privileges, the *setuid()* function sets the real user ID, effective user ID, and the saved set-user-ID to *uid*.

If the process does not have appropriate privileges, but *uid* is equal to the real user ID or the saved set-user-ID , the *setuid()* function sets the effective user ID to *uid*; the real user ID and saved set-user-ID remain unchanged by this functioncall.

Otherwise:

If the process has appropriate privileges, the *setuid()* function sets the real user ID and effective user ID to *uid*.

If the process does not have appropriate privileges, but *uid* is equal to the real user ID, the *setuid()* function sets the effective user ID to *uid*; the real user ID remains unchanged by this function call.

If the process does not have appropriate privileges, but *gid* is equal to the real group ID, the *setgid()* functionsets the effective group ID to *gid*; the real group ID remains unchanged by this function call.

Implementation Considerations

None.

Errors

If any of the following conditions occur, the *setuid()* function shall return -1 and set `errno` to the corresponding value:

`EINVAL`	CAUSE	The signal *sig* is not a valid signal number, or *pid* is -1.
	ACTION	Refer to Table 3-5 for descriptions of valid signal numbers, or set *pid* to a valid value.
`EPERM`	CAUSE	The caller does not have permission to send the signal to any receiving process.
	ACTION	Refer to the `kill()` function description for signal permission rules.

If any of the following conditions occur, the *setgid()* function shall return -1 and set `errno` to the corresponding value:

`EINVAL`	CAUSE	The signal *sig* is not a valid signal number, or *pid* is -1.
	ACTION	Refer to Table 3-5 for descriptions of valid signal numbers, or set *pid* to a valid value.
`EPERM`	CAUSE	The caller does not have permission to send the signal to any receiving process.
	ACTION	Refer to the `kill()` function description for signal permission rules.

See Also

`exec()`, `getuid()`, POSIX.1 (Section 3.3.2).

sigaction

Examines and/or changes a signal action.

Syntax

```
#include <signal.h>
int sigaction (int sig, const struct sigaction *act,
 struct sigaction *oact);
```

Parameters

sig The signal number of the signal to examine or change. Valid signals are described in Table 3-5.

act If not NULL, a pointer to a structure of type `sigaction` that describes a new signal action to be associated with *sig*. If NULL, the current signal action is unchanged.

oact If not NULL, a pointer to a structure of type `sigaction` that returns the description of the current action for the signal *sig* (prior to any changes). If NULL, the current action is not returned.

Return Values

0 Success.

-1 An error occurred. The signal action is not changed, and `errno` is set to indicate the error condition.

Description

The `sigaction()` function enables the calling process to examine or change (or both) the action associated with the specified signal.

In order to examine the current action associated with a signal without changing the current action, set *act* to NULL. In order to change an action associated with a signal, define the new signal action in a structure of type `sigaction` and pass it in *act*. Refer to the following discussion of the `sigaction` structure.

The `sigaction` structure, defined in `<signal.h>`, includes the following fields:

Member Type	Member Name	Description
`void (*)()`	*sa_handler*	Either `SIG_DFL` for the default action, `SIG_IGN` to ignore the signal, or a pointer to a signal handling function (a signal handler).
`sigset_t`	*sa_mask*;	Additional signals to be blocked during execution of the signal handler specified in *sa_handler*.
`int`	*sa_flags*;	If *sig* specifies `SIGCHLD` and *sa_flags* specifies `SA_NOCLDSTOP`, a `SIGCHLD` signal is not generated for the calling process whenever any of its child processes stop. If *sig* specifies `SIGCHLD` and *sa_flags* does not specify `SA_NOCLDSTOP`, `SIGCHLD` is generated for the calling process whenever any of its child processes stop.

When installing a new signal handler, you must specify in *sa_mask* any additional signals to be blocked during the execution of the signal handler. When a signal is caught by a signal handler installed by `sigaction()`, a new signal mask is calculated and installed for the duration of the signal handler (or until a call to either `sigprocmask()` or `sigsuspend()`).

This mask is formed by taking a union of the current signal mask and *sa_mask* for the signal being delivered, then including the signal being delivered. If and when the signal handler returns normally, the original signal mask is restored.

The signals `SIGKILL` and `SIGSTOP`, if specified in the *sa_mask* field, are removed by the system without error.

The structure passed in *sa_mask* must be initialized by either the `sigemptyset()` or the `sigfillset()` functions before adding or removing signals from it using the `sigaddset()` or `sigdelset()` functions.

A signal action installed by `sigaction()` remains in effect until changed by another call to `sigaction()` or until the next call to one of the `exec()` functions.

The `sigaction()` function is incompatible with the ANSI C `signal()` function. The `sigaction()` function can return and reinstall a signal action that was originally installed by `signal()`; however, the structure that `sigaction()` returns in *oact* may not reliably be examined by the caller. If this same signal action is later reinstalled, without modification, by another call to `sigaction()`, the result is as if the original call to `signal()` were repeated.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a bad address in attempting to use the *act* or *oact* parameters.
	ACTION	Make sure that the pointer is correctly initialized.
`EINVAL`	CAUSE	One of the following: ■ The *sig* parameter is not a valid signal number. ■ An attempt was made to handle a signal that cannot be handled or to ignore a signal that cannot be ignored. ■ An attempt was made to change the action from `SIGDFL` for a signal that cannot be handled or ignored.
	ACTION	Refer to Table 3-5 for descriptions of valid and supported signal numbers.

See Also

`kill()`, `sigprocmask()`, `sigsuspend()`, `<signal.h>`, POSIX.1 (Section 3.3.4)

sigaddset

Adds a signal to a signal set.

Syntax

```
#include <signal.h>
int sigaddset (sigset_t *set, int sig);
```

Parameters

set A pointer to a structure of type `sigset_t` containing a set of signals to which *sig* is to be added.

sig The signal number of the signal to add to *set*. Valid signals are described in Table 3-5.

Return Values

0 Success.

-1 An error occurred. The signal is not added, and `errno` is set to indicate the error condition.

Description

The `sigaddset()` function adds the signal *sig* to the set of signals specified in the structure pointed to by *set*.

The structure of type `sigset_t` pointed to by *set* must be initialized by `sigemptyset()` or `sigfillset()` prior to being used by `sigaddset()`; otherwise, the results are undefined.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *set* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINVAL`	CAUSE	The signal *sig* is not a valid signal number.
	ACTION	Refer to Table 3-5 for descriptions of valid signal numbers.

See Also

`sigaction()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `<signal.h>`, POSIX.1 (Section 3.3.3)

sigdelset

Deletes a signal from a signal set.

Syntax

```
#include <signal.h>
int sigdelset (sigset_t *set, int sig);
```

Parameters

set A pointer to a structure of type `sigset_t` containing a set of signals from which *sig* is to be deleted.

sig The signal number of the signal to delete from *set*. Valid signals are described in Table 3-5.

Return Values

0 Success.

-1 An error occurred. The signal is not deleted, and `errno` is set to indicate the error condition.

Description

The `sigdelset()` function deletes the signal *sig* from the set of signals specified in the structure pointed to by *set*.

The structure of type `sigset_t` pointed to by *set* must be initialized by `sigemptyset()` or `sigfillset()` prior to being used by `sigdelset()`; otherwise, the results are undefined.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *set* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINVAL`	CAUSE	The signal *sig* is not a valid signal number.
	ACTION	Refer to Table 3-5 for descriptions of valid signal numbers.

See Also

`sigaction()`, `sigaddset()`, `sigemptyset()`, `sigfillset()`, `sigismember()`, `<signal.h>`, POSIX.1 (Section 3.3.3)

sigemptyset

Initializes a signal set to the empty set.

Syntax

```
#include <signal.h>
int sigemptyset (sigset_t *set);
```

Parameters

set A pointer to a structure of type `sigset_t` to initialize to the empty set.

Return Values

0 Success.

-1 An error occurred. The signal set is not initialized, and `errno` is set to indicate the error condition.

Description

The `sigemptyset()` function initializes *set* to the empty set. All signals described in Table 3-5 are excluded from the set.

The `sigemptyset()` or `sigfillset()` function must be called to initialize the structure of type `sigset_t` pointed to by *set* prior to its use by other functions.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to the following value:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *set* parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`sigaction()`, `sigaddset()`, `sigdelset()`, `sigfillset()`, `sigismember()`, `<signal.h>`, POSIX.1 Section 3.3.3.

sigfillset

Initializes a signal set to the full set.

Syntax

```
#include <signal.h>
int sigfillset (sigset_t *set);
```

Parameters

set	A pointer to a structure of type `sigset_t` to initialize to the full set.

Return Values

0	Success.
-1	An error occurred. The signal set is not initialized, and `errno` is set to indicate the error condition.

Description

The `sigfillset()` function initializes *set* to the full set. All signals described in Table 3-5 are included in the set.

The `sigfillset()` or `sigemptyset()` function must be called to initialize the structure of type `sigset_t` pointed to by *set* prior to its use by other functions.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to the following value:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *set* parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigismember()`, `<signal.h>`, POSIX.1 (Section 3.3.3)

sigismember

Tests whether a signal is a member of a signal set.

Syntax

```
#include <signal.h>
int sigismember (const sigset_t *set, int sig);
```

Parameters

set A pointer to a structure of type `sigset_t` containing a set of signals to test.

sig The signal number of the signal to test for membership in *set*. Valid signals are described in Table 3-5.

Return Values

1 The signal *sig* is a member of the signal set.

0 The signal *sig* is not a member of the signal set.

-1 An error occurred. The test is not performed, and `errno` is set to indicate the error condition.

Description

The `sigismember()` function tests whether or not the signal *sig* is a member of the set of signals specified in the structure pointed to by *set*.

Implementation Considerations

Refer to the `EFAULT` error description below.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *set* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINVAL`	CAUSE	The signal *sig* is not a valid signal number.
	ACTION	Refer to Table 3-5 for descriptions of valid signal numbers.

See Also

`sigaction()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()`, `<signal.h>`, POSIX.1 (Section 3.3.3)

siglongjmp

Restore an environment previously saved by `sigsetjmp()`.

Syntax

```
#include <setjmp.h>
void siglongjmp (sigjmp_buf env, int val);
```

Parameters

env Passes information needed to restore a previous environment. This variable was used in a previous call to `sigsetjmp()` to save the environment. The type `sigjmp_buf` (defined in `<setjmp.h>`) defines an array of unsigned integers. For this reason, the *env* argument does not require an `&` operator..

val Passes a value to be returned by `sigsetjmp()`. If a zero is passed in this argument, it is changed to a value of 1 to ensure that `siglongjmp()` never causes `sigsetjmp()` to return a zero value.

Return Values

None.

Description

The `siglongjmp()` function restores the environment saved in *env* by a previous call to the `sigsetjmp()` macro. If the *env* argument was initialized by a call to `sigsetjmp()` with a non-zero value passed in the *savemask* argument, the saved signal mask is also restored by `siglongjmp()`. If the *env* argument was not initialized by a call to `sigsetjmp()`, the operation of `siglongjmp()` is undefined.

After `siglongjmp()` is completed, the program executes as if the call to `sigsetjmp()` (which stored information into the *env* argument) had returned a second time. In this case, `sigsetjmp()` returns either the non-zero value passed in the *val* argument of `siglongjmp()` or 1 if zero was passed in *val*.

The restoration of the environment includes trimming the stack so that all stack frames beyond the frame marked by *env* are removed. The `siglongjmp()` function cannot add stack frames. This means that if a sequence of functions is:

```
A == calls ==> B == calls ==> C
```

and `sigsetjmp()` is used in function C to save an environment in a global *env*, functions B or A may not contain any `siglongjmp()` calls that reference the *env* values. Only subordinate functions may issue calls to `siglongjmp()`. As a special case, a function may issue a `siglongjmp()` call that references a `sigsetjmp()` within itself, although this is not usually done.

The values of objects of automatic storage duration that are not qualified by `volatile` are indeterminate if they have changed since the call to `sigsetjmp()`.

The `siglongjmp()` function will work correctly in the context of signals and interrupts and any of their associated functions. However, if the `siglongjmp()` function is invoked from a nested signal handler, the operation of `siglongjmp()` is undefined.

See Also

`sigsetjmp()`, POSIX 1003.1 Section 8.3.1

sigpending

Returns the set of pending signals.

Syntax

```
#include <signal.h>
int sigpending (sigset_t *set);
```

Parameters

set A pointer to a structure of type `sigset_t` that is to contain the signals that are blocked from delivery and pending for the calling process.

Return Values

0 Success.

-1 An error occurred. No information is returned, and `errno` is set to indicate the error condition.

Description

The `sigpending()` function returns to *set* the set of signals that are blocked from delivery and pending for the calling process.

Implementation Considerations

Refer to the `EFAULT` error description below.

Signals that are both blocked and ignored for the calling process remain pending if generated for the process.

Errors

If an error occurs, `errno` is set to the following value:

`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *set* parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`sigprocmask()`, `<signal.h>`, POSIX.1 (Section 3.3.6)

sigprocmask

Examines or changes blocked signals.

Syntax

```
#include <signal.h>
int sigprocmask (int how, const sigset_t *set,
 sigset_t *oset);
```

Parameters

how Indicates how the process's signal mask should be changed by the parameter *set*. One only of the following values must be passed:

`SIG_BLOCK`	Add the specified signals in *set* to the process's signal mask.
`SIG_UNBLOCK`	Delete the specified signals in *set* from the process's signal mask.
`SIG_SETMASK`	Replace the process's signal mask with the signal mask pointed to by *set*.

set If not NULL, a pointer to a structure of type `sigset_t` containing a set of signals to use when changing the calling process's signal mask in the manner defined by *how*. If NULL, *how* is ignored and the process's signal mask remains unchanged.

oset If not NULL, a pointer to a structure of `sigset_t` that returns the process's previous signal mask (prior to any changes). If NULL, the current signal mask is not returned.

Return Values

0 Success.

-1 An error occurred. The process's signal mask is not changed, and `errno` is set to indicate the error condition.

Description

The `sigprocmask()` function allows the caller to examine or change (or both) the calling process's signal mask. If any pending unblocked signals remain after a call to `sigprocmask()`, at least one of those signals is delivered to the calling process before the function returns.

It is not possible to block the signals `SIGKILL` and `SIGSTOP`. If specified in the structure pointed to by *set*, they are removed by the system without error.

Implementation Considerations

Refer to the `EFAULT` error description below.

Refer to Table 3-5 for implementation considerations associated with signals.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a bad address in attempting to use the *set* or *oset* parameters.
	ACTION	Make sure that the pointer is correctly initialized.
`EINVAL`	CAUSE	The value of *how* is not valid.
	ACTION	Specify valid values for *how*.

See Also

`sigaction()`, `sigpending()`, `sigsuspend()`, `<signal.h>`, POSIX.1 (Section 3.3.5)

sigsetjmp

Save the current environment and signal mask.

Syntax

```
#include <setjmp.h>
int sigsetjmp (sigjmp_buf env, int savemask);
```

Parameters

env Returns the current environment for later use in a call to `siglongjmp()`. If *savemask* is set to a non-zero value, the current signal mask is also returned in *env*. The type `sigjmp_buf` (defined in `<setjmp.h>`) defines an array of unsigned integers. For this reason, the *env* argument does not require an `&` operator.

savemask If a non-zero value is passed in *savemask*, the current signal mask is returned in *env*. If zero is passed, the current signal mask is not saved.

Description

A call to `setsetjmp()` creates an entry point in a program that can be accessed via `siglongjmp()`. The `sigsetjmp()` macro saves the current environment of the calling process in *env*. If *savemask* is set to a non-zero value, the current signal mask is also saved in *env*. A subsequent call to `siglongjmp()` requires that the *env* variable be passed to restore the environment.

If a zero value is returned, the return is from `sigsetjmp()` itself and not a return as a result of a call to `siglongjmp()`.

If a nonzero value is returned, the return is a result of a call to `siglongjmp()`. After `siglongjmp()` is completed, the program executes as if the call to `sigsetjmp()` had returned a second time. In this case, `sigsetjmp()` returns either the non-zero value passed in the *val* argument of `siglongjmp()` or 1 if zero was passed in *val*.

Return Values

0	Successful completion of a call to `sigsetjmp()`.
<>0	Successful completion of a call to `siglongjmp()`. The value is that of the *val* parameter passed to `siglongjmp()`, or 1 if a zero was passed in the *val* parameter.

See Also

`siglongjmp()`, POSIX 1003.1 (Section 8.3.1)

sigsuspend

Replaces the calling process's signal mask and suspends the calling process to wait for a signal.

Syntax

```
#include <signal.h>
int sigsuspend (sigset_t *sigmask);
```

Parameters

sigmask If not NULL, a pointer to a structure of type `sigset_t` that contains a new signal mask to be installed before suspending the calling process. If NULL, the process's current signal mask is used.

Return Values

No return	Because `sigsuspend()` suspends process execution indefinitely, there is no return value indicating success.
-1	An error occurred, and `errno` is set to indicate the error condition.

Description

The `sigsuspend()` function replaces the calling process's signal mask with the set of signals pointed to by *sigmask*. It then suspends the process until the delivery of a signal whose action is either to execute a signal-handling function (signal handler) or to terminate the process.

If the action is to execute a signal handler, upon completion of the signal handler, `sigsuspend()` returns and restores the process's previous signal mask. If the signal action is to terminate the process, `sigsuspend()` does not return.

It is not possible to block the signals `SIGKILL` and `SIGSTOP`. If specified in the structure pointed to by *sigmask*, they are removed by the system without error.

Implementation Considerations

Refer to the `EFAULT` error description below.

If the *sigmask* parameter of the `sigsuspend()` function is set to NULL, the process is suspended with the current signal mask. This implementation is considered an extension to the POSIX.1 standard. A strictly conforming POSIX application should pass in the *sigmask* parameter of the `sigsuspend()` function the current signal mask returned by a successful call to `sigprocmask()` where *set* is set to NULL.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a bad address in attempting to use the *sigmask* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINTR`	CAUSE	A signal was caught by the process, and control was returned from a signal-handling function.
	ACTION	No action required.

See Also

`pause()`, `sigaction()`, `sigpending()`, `sigprocmask()`, `<signal.h>`, POSIX.1 (Section 3.3.7)

sleep

Delays process execution.

Syntax

```
#include <unistd.h>
unsigned int sleep (unsigned int seconds);
```

Parameters

seconds	Specifies the number of real time seconds to sleep.

Return Values

No return	If the action associated with a signal is to terminate, `sleep()` does not return.
0	The requested time has elapsed.
>0	The difference between *seconds* and the actual number of seconds slept before delivery of a signal whose action is to execute a signal handling function.

Description

The `sleep()` function suspends the calling process from execution either for the number of real-time seconds specified by the *seconds* parameter or until the delivery of a signal whose action is either to execute a signal-handling function or to terminate the process.

If *seconds* real-time seconds have passed without receipt of a signal with the appropriate action, `sleep()` returns control to the calling process.

If the action associated with a received signal is to execute a signal handling function, upon completion of the function, `sleep()` returns control to the calling process. If the signal action is to terminate the process, `sleep()` does not return.

Due to system activity, the process may be suspended for more than the number of real-time seconds indicated by the *seconds* parameter.

Implementation Considerations

The `SIGALRM` signal is not used to implement `sleep()`.

Errors

None.

See Also

`alarm()`, `pause()`, `sigaction()`, POSIX.1 (Section 3.4.3).

stat

Returns file status information.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buffer);
```

Parameters

pathname A pointer to a string containing a pathname of the file or directory from which to obtain information. The pathname must be terminated by a null character.

buffer A pointer to a buffer of type `struct stat` (defined in `<sys/stat.h>`) where file status information is returned.

Return Values

0 Success.

-1 An error occurred. File status information is not returned and `errno` is set to indicate the error condition.

Description

The `stat()` function returns status information on the specified file or directory to the structure pointed to by *buffer*.

The `stat()` function updates to the current time all time fields that have been previously marked for update. All update marks are removed.

Implementation Considerations

Refer to the EACCES, EFAULT, EIMPL, and ESYSERR error descriptions below.

Access permissions to the file are not required, but if the file or directory has an MPE/iX ACD, the calling process must have MPE/iX read ACD (RACD) access to the file or directory, or an error occurs.

Errors

If an error occurs, errno is set to one of the following values:

EACCES	CAUSE	The calling process does not have search permission to a component of the pathname.
	ACTION	Make sure that the calling process has search permission to all components of the pathname.
EFAULT	CAUSE	The system detected a NULL or bad address in attempting to use the *buffer* or *pathname* parameters, or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
ENAMETOOLONG	CAUSE	One of the following: ■ The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>). ■ A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.
ENOENT	CAUSE	The specified file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
ENOTDIR	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.

`EPERM`	CAUSE	The calling process does not have MPE/iX read ACD (RACD) access to the file, or the pathname begins with two slash characters (//).
	ACTION	Make sure that the calling process has RACD access to the file before calling **`stat()`**, or do not begin the pathname with two slash characters (//).
`ESYSERR`	CAUSE	Access denied. Unable to map UID and GID to owner of the file or directory either because user database is corrupted or because the MPE/iX file type is not supported by POSIX/iX library functions.
	ACTION	Check to see if the user database is corrupted, or if the MPE/iX file type is supported by POSIX/iX library functions.

See Also

`creat()`, `dup()`, `fstat()`, `open()`, `<sys/stat.h>`, POSIX.1 (Section 5.6.2)

symlink

Creates a symbolic link to a file.

Syntax

```
#include <unistd.h>
init symlink(const char *pname, const char *slink);
```

Parameters

pname Is the pathname contained in the symbolic link.

slink Is the name of the symbolic link created.

Return Values

Upon successful completion, the *symlink()* function will return zero. Otherwise, a value of -1 will be returned and `errno` will be set to indicate the error.

Description

The *symlink* function will create a symbolic link called *slink*, that contains the pathname specified by *pname* (*slink* is the name of the symbolic link created, *pname* is the pathname contained in the symbolic link).

Implementation Considerations

None.

Errors

If an error occurs, errno is set to one of the following values:

EACCES	CAUSE	The calling process does not have search permission to a component of the pathname.
	ACTION	Make sure that the calling process has search permission to all components of the pathname.
ENAMETOOLONG	CAUSE	One of the following: ■ The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>). ■ A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.
EEXIST	CAUSE	The filenamed by *slink* already exists.
	ACTION	Make sure the *slink* does not exist.
ELOOP	CAUSE	A loop exists in symbolic links encountered during resolution of the *slink* argument. This error may be returned if more than {POSIX_SYMLOOP} symbolic links are encountered during resolution of the *slink* argument.
	ACTION	Make sure that there is not a loop in the symbolic links that loops more than POSIX_SYMLOOP.
ENOTDIR	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
ENOSPC	CAUSE	The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.
	ACTION	Create the *slink* on a writable volume (file system).
EROFS	CAUSE	The file *slink* would reside on a read-only file system.
	ACTION	Create the *slink* on a writable volume (file system).

See Also

link(), readlink(), unlink()

sysconf

Determine system configuration options.

```
#include <unistd.h>
long sysconf(int name);
```

Parameters

name	specifies the system configuration option for which you want to obtain the value. The value of *name* is given may be any one of a set of symbols defined in <unistd.h>; each of these symbols corresponds to an environment variable or manifest constant which gives a system configuration option.

Return Values

`sysconf()` returns the value associated with the specified *name*. If *name* is not recognized, or stands for a symbol which is undefined, then `sysconf()` returns -1.

Description

The name argument may be any one of the following symbols:

`_SC_ARG_MAX`	Stands for `ARG_MAX` defined in <limits.h>—the maximum number of bytes of arguments and environment data that can be passed in an `exec()` call.
`_SC_BC_BASE_MAX`	Stands for `BC_BASE_MAX` defined in <unistd.h>—the maximum value for *ibase* and *obase* in `bc(1)`.
`_SC_BC_DIM_MAX`	Stands for `BC_DIM_MAX` defined in <unistd.h>—the maximum number of elements in a `bc(1)` array.
`_SC_BC_SCALE_MAX`	Stands for `BC_SCALE_MAX` defined in <unistd.h>—the maximum `scale` in `bc(1)`.

`_SC_BC_STRING_MAX`	Stands for `BC_STRING_MAX` defined in <unistd.h>—the maximum length of a string accepted by `bc(1)`.
`_SC_CHILD_MAX`	Stands for `CHILD_MAX` defined in <limits.h>—the maximum number of processes that a real user ID may have executing simultaneously.
`_SC_CLK_TCK`	Stands for `CLK_TCK` defined in <time.h>—the number of clock ticks in a second.
`_SC_COLL_WEIGHTS_MAX`	Stands for `COLL_WEIGHTS_MAX` defined in <unistd.h>—the maximum number of weights that can be assigned to an entry of the `LC_COLLATE` order keyword in the locale definition file.
`_SC_EXPR_NEST_MAX`	Stands for `EXPR_NEST_MAX` defined in <unistd.h>—the largest number of expressions that can be nested within parentheses by `expr(1)`.
`_SC_JOB_CONTROL`	Stands for `_POSIX_JOB_CONTROL` which may be defined in <unistd.h>—this indicates that certain job control operations are implemented by this version of the operating system. If `_POSIX_JOB_CONTROL` is defined, various functions (for example, `setpgid()`) have greater functionality than when it is not defined.
`_SC_LINE_MAX`	Stands for `LINE_MAX` defined in <unistd.h>—the maximum length of a utility's input line when the utility processes text files. This length includes the newline on the end of the line.
`_SC_NGROUPS_MAX`	Stands for `NGROUPS_MAX` defined in <limits.h>—the maximum number of supplementary group IDs that may be associated with a process.

_SC_OPEN_MAX	Stands for OPEN_MAX defined in <limits.h>—the maximum number of files that a single process may have open at one time.
_SC_RE_DUP_MAX	Stands for RE_DUP_MAX defined in <unistd.h>—the largest number of repeated occurrences of a regular expression that you can use in the notation \{m,n\}.
_SC_SAVED_IDS	Stands for _POSIX_SAVED_IDS which may be defined in <unistd.h>—this indicates that this POSIX implementation has a saved set-user-ID and a saved set- group-ID. This affects the behavior of functions like setuid() and setgid().
_SC_STREAM_MAX	Stands for _POSIX_STREAM_MAX which may be defined in <limits.h>—the number of streams that one process can have open at one time.
_SC_TZNAME_MAX	Stands for _POSIX_TZNAME_MAX which may be defined in <limits.h>—the maximum number of bytes supported for the name of a time zone (not of the TZ variable).
_SC_VERSION	Stands for _POSIX_VERSION which may be defined in <unistd.h>—this indicates the version of the POSIX.1 standard to which the system conforms.
_SC_2_C_BIND	Stands for _POSIX2_C_BIND which may be defined in <unistd.h>—if this is defined, the system supports the C Language Bindings Option of POSIX.2.
_SC_2_C_DEV	Stands for _POSIX2_C_DEV which may be defined in <unistd.h>—if this is defined, the system supports the C Language Development Utilities Option of POSIX.2.

`_SC_2_CHAR_TERM`	Stands for `_POSIX2_CHAR_TERM` which may be defined in <unistd.h>—if this is defined, the system supports at least one terminal type capable of all operations necessary for the User Portability Utilities. This is only on if `_SC_2_UPE` is on.
`_SC_2_FORT_DEV`	Stands for `_POSIX2_FORT_DEV` which may be defined in <unistd.h>—if this is defined, the system supports the FORTRAN Development Utilities Option of POSIX.2.
`_SC_2_FORT_RUN`	Stands for `_POSIX2_FORT_RUN` which may be defined in <unistd.h>—if this is defined, the system supports the FORTRAN Runtime Utilities Option of POSIX.2.
`_SC_2_LOCALEDEF`	Stands for `_POSIX2_LOCALEDEF` which may be defined in <unistd.h>—if this is defined, the system supports the creation of locales.
`_SC_2_SW_DEV;`	Stands for `_POSIX2_SW_DEV` which may be defined in <unistd.h>—if this is defined, the system supports the Software Development Utilities Option of POSIX.2.
`_SC_2_UPE`	Stands for `_POSIX2_UPE` which may be defined in <unistd.h>—if this is defined, the system supports the User Portability Utilities Option.
`_SC_2_VERSION`	Stands for `_POSIX2_VERSION` which may be defined in <unistd.h>—this indicates the version of the POSIX.2 standard to which the system conforms.

Errors

If `sysconf()` fails to recognize name, it returns -1 and sets errno to the value:

`EINVAL`	CAUSE	The value specified for the name argument was invalid.
	ACTION	Specify a valid value for name.

See Also

`bc(1)`, `expr(1)`, `paste(1)`

system

Execute a command using the shell.

Syntax

```
#include <stdlib.h>
int system(const char *command);
```

Parameters

command is a string giving the command line for the command you want to execute using the shell (MPE/iX Shell).

Return Values

If command is NULL, `system()` returns -1.

If command is not NULL, `system()` returns the exit status of the `sh` command that executes command. If sh cannot be invoked to execute the command, the return value of `system()` is the value that would be received if `sh` terminated with exit(127).

If `system()` cannot `fork()` a child process, it returns -1 and sets `errno` to an appropriate value. It uses the same `errno` values used by `fork()` for its possible failures.

Description

`system()` executes the command specified by *command*. It does this as if it spawns a child process with `fork()`, then the child process invokes the shell `sh` with

```
execl(shellpath, "sh", "-c", command, NULL);
```

where *shellpath* is the pathname of the file that contains the MPE/iX Shell.

`system()` ignores the `SIGINT` and `SIGQUIT` signals while waiting for the command to terminate. It also blocks the `SIGCHLD` signal. After the command terminates, the calling process can examine the return value from `system()` to determine if any of these signals should be handled.

Implementation Considerations

None.

Errors

None.

See Also

sh(1)

time

Returns the number of seconds since the Epoch.

Syntax

```
#include <time.h>
time_t time (time_t *tloc);
```

Parameters

tloc If not NULL, a pointer to a variable of type `time_t` where the number of seconds since the Epoch is returned. If NULL, no value is stored.

Return Values

>=0 Success.

-1 An error occurred. The time is not returned, and `errno` is set to indicate the error condition.

Description

The `time()` function calculates and returns the number of seconds since the last Epoch (00:00:00 Coordinated Universal Time (UTC) January 1, 1970).

If *tloc* is not NULL, the same value is returned in the variable pointed to by *tloc*. If *tloc* is NULL, no value is returned in *tloc*.

Implementation Considerations

Refer to the `EFAULT` error description below.

The `TZ` environment variable does not affect this function.

Errors

If an error occurs, `errno` is set to the following value:

`EFAULT`	CAUSE	The system detected a bad address in attempting to use the *tloc* parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`<time.h>`, POSIX.1 (Section 4.5.1)

times

Gets process times.

Syntax

```
#include <sys/times.h>
clock_t (struct tms *buffer);
```

Parameters

buffer insert parameter info here

Return Values

Upon successful completion, *times()* will return the elapsed real time, in clock ticks, since and arbitrary point in the past (for example, system start-up time). This point does not change from one invocation of *times()* within the process to another. The return value may overflow the possible range of type *clock_t*. The *times* function fails, a value of (`clock_t`)-1 is returned and `errno` is set to indicate the error.

Description

The *times()* function will fill the structure pointed to by *buffer* with time-accounting information. The type *clock_t* and the *tms* structure are defined in <sys/times.h>; the *tms* structure will contain at least the following members:

Member Type	Member Name	Description
clock_t	*tms_utime*	User CPU time.
clock_t	*tms_stime*	System CPU time.
clock_t	*tms_cutime*	User CPU time of terminated child processes.
clock_t	*tms_cstime*	System CPU time of terminated child processes.

All times are measured in terms of the number of clock ticks used.

The times of a terminated child process are included in the *tms_cutime* and *tms_cstime* elements of the parent when a *wait()* or *waidpid()* function returns the process ID of this terminated child. See 3.2.1. If a child process has not waited for its terminated children, their times will not be included in its times.

The value *tms_utime* is the CPU time charged for the execution of user instructions.

The value *tms_stime* is the CPU time charged for execution by the system on behalf of the process.

The value *tms_cstime* is the sum of the *tms_stimes* and *tms_cstimes* fo the child processes.

Implementation Considerations

Because MPE/iX tracks the CPU time used by a process as a single total value, and does not distinguidsh between "user" and "system" CPU time, the information returned in the *tms* structure is based on estimated percentages of the total CPU time. The sum of the *tms_utime* and *tms_stime* fields will still reflect the total CPU time of the process. This is similar for the terminated child times calculation.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	The system detected a NULL or bad pointer in attempting to use the buffer parameter.
	ACTION	Make sure that the pointer is correctly initialized.

See Also

`exec()`, `fork()`, `sysconf()`, `time()`, `wait()`, POSIX.1

ttyname

Determines terminal device name.

Syntax

```
#include <unistd.h>
char *ttyname(int fildes);
```

Parameters

fildes   An open file descriptor.

Return Values

The *ttyname()* function returns a *NULL* pointer if *filedes* is not a valid file descriptor associated with a terminal or if the pathname cannot be determined.

Description

The *ttyname* function returns a pointer to a string containing a null-terminated pathname of the terminal associated with file descriptor *fildes*.

Implementation Considerations

None.

Errors

If an error occurs, `errno` is set to the following value:

`EBADF`	CAUSE	The *fildes* parameter is not a valid open file descriptor.
	ACTION	Check to see if *fildes* has been altered or not initialized.

See Also

`ctermid()`, `isatty()`, POSIX.1

umask

Sets a process's file mode creation mask.

Syntax

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask (mode_t cmask);
```

Parameters

cmask	A bit map specifying a creation mask. Bits that are not access permission bits must be set to zero or an error occurs.

Return Values

0	Success. This is the first time `umask()` has been invoked by the calling process. There is no previous valid file creation mask.
-1	An error occurred, and `errno` is set to indicate the error condition.
Any other value	Success. The previous file creation mask of the calling process.

Description

The `umask()` function sets the calling process's file mode creation mask to the mask specified in the object of type `mode_t` passed in *cmask* and returns the previous value of the mask.

Only the file permission bits (defined in `<sys/stat.h>`) of *cmask* are used. Bits of *cmask* that are not file permission bits must contain zeros or an error occurs.

The process's file mode creation mask is used during `open()`, `create()`, and `mkdir()` calls to turn off permission bits in the *mode* parameter supplied. Bit positions that are set in *cmask* are cleared in the mode of the created file.

Implementation Considerations

Refer to the EIMPL error description below.

The first time umask() is invoked by a process, zero is returned to indicate that the previous file creation mask was not initialized.

Errors

If an error occurs, errno is set to the following value:

EIMPL	CAUSE	Bits of *cmask* that are not file permission bits do not contain zeros.
	ACTION	Make sure that bits that are not file permission bits contain zeros.

See Also

chmod(), creat(), mkdir(), open(), <sys/stat.h>, POSIX.1 (Section 5.3.3)

uname

Returns current system ID's.

Syntax

```
#include <sys/utsname.h>
init uname(struct utsname *name);
```

Parameters

name	A pointer to a string of characters that will return system identification.

Return Values

>=0	Successful completion.
-1	Error and `errno` is set to indicate the error.

Description

The *uname()* function stores information identifying the current operating system in the utsname structure pointed to by the argument *name*.

The structure *utsname* is defined in the header <sys/utsname.h> and contains at least the members shown below:

sysname	Name of this implementation of the operating system.
nodename	Name of this node within an implementation-specified communications network.
release	Current operation system release ID.
version	Current operation system version ID.
machine	Name of the hardware type on which the system is running.

Each of these data items is a null-terminated array of *char*.

The inclusion of the *nodename* member in this structure does not imply that it is sufficient information for interfacing to communications networks.

A sample output of this parameter displays as follow:

```
sysname = MPE/iX
nodename = STARS.ITG.HP
release = A.41.00
version = A.51.07
machine = SERIES 955
```

Implementation Considerations

The node name is retrieved from NMCONFIG.PUB.SYS and is not necessarily sufficient information for interfacing to communications networks. The release ID is the manufacture release ID, known as the release vuf on MPE/iX. The version ID stands for the version of the MPE/iX OS product.

Since the POSIX standard does not specify any error conditions that are required to be detected for the *uname()* function, all the error conditions are implementation defined. Successful completion will have a function return of zero.

Errors

If an error occurs, `errno` is set to one of the following values:

`EFAULT`	CAUSE	A null or bad address was detected in attempting to use the structure pointed to by the *name* argument.
	ACTION	Check to see if the pointer is initialized and/or the structure is defined correctly.
`ESYSERR`	CAUSE	An internal operating system error has occurred; an error not directly applicable to the POSIX.1 functionality.
	ACTION	Contact Hewlett-Packard for support.

See Also

`exec()`, `getuid()`, POSIX.1 (Section 3.3.2).

unlink

Removes a link from a file.

Syntax

```
#include <unistd.h>
int unlink (const char *pathname);
```

Parameters

pathname A pointer to a string containing the pathname of a file to unlink (purge). The pathname must be terminated by a null character.

Return Values

0	Success.
-1	An error occurred. The file is not unlinked, and `errno` is set to indicate the error condition.

Description

The `unlink()` function removes the link name specified by *pathname*. It removes the filename pointed to by *pathname* from the parent directory, then decrements the file link count. When the link count of the file becomes zero and no process has the file open, the file is purged from the system and is no longer accessible.

If one or more processes have the file open when the link count becomes zero, the file is not purged until all references to the file have been closed.

Upon successful completion, `unlink()` marks for update the `st_ctime` and `st_mtime` time fields of the parent directory.

Implementation Considerations

Refer to the EFAULT, EIMPL, EPERM, and ESYSERR error descriptions below.

POSIX/iX does not support using `unlink()` on directories. Instead, use `rmdir()` to remove a directory.

POSIX/iX does not support multiple hard links to files or soft links to files or directories.

Every file has a link count of 1 when created. Files being unlinked cause the link count of the file to be decremented from 1 to 0.

Errors

If an error occurs, `errno` is set to one of the following values:

EACCES	CAUSE	The calling process either does not have search permission to a component of the pathname or does not have write permission to the parent directory.
	ACTION	Make sure that the calling process has search permission for all components of the pathname and write permission to the parent directory.
EFAULT	CAUSE	The system detected a NULL or bad address in attempting to use the *pathname* parameter or the pathname was not terminated by a null character.
	ACTION	Make sure that the pointer is correctly initialized.
EIMPL	CAUSE	The pathname begins with two slash characters (//).
	ACTION	Do not begin pathnames with two slash characters (//).
ENAMETOOLONG	CAUSE	One of the following: ■ The length of the pathname exceeds the {PATH_MAX} limit (defined in the file <limits.h>). ■ A component of the pathname is longer than {NAME_MAX} (defined in <limits.h>), and {_POSIX_NO_TRUNC} is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the {NAME_MAX} or {PATH_MAX} limits.

`ENOENT`	CAUSE	The specified file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`EPERM`	CAUSE	The specified file is a directory.
	ACTION	Do not attempt to unlink a directory. Use **`rmdir()`** instead.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`close()`, `open()`, `rmdir()`, POSIX.1 (Section 5.5.1)

utime

Sets access and modification times of a file.

Syntax

```
#include <sys/types.h>
#include <utime.h>
int utime(const char *path, const struct utimbuf *times),
```

Parameters

path A pointer to a character array containing the pathname of the file that is to become the new process image. The pathname must be terminated by a null character.

times If not NULL, a pointer to a *utimbuf* structure containing the access and modification times. Only the owner of the file, the system manager, or the account manager can use the *utime* function this way.

If this argument is NULL, however, the access and modification time of the file are set to the current time.

Return Values

0 Successful completion.

-1 Error. `errno` is set to indicate the error, and the file times shall not be affected.

Description

The *utime()* function sets the access and modification times of the named file.

If the *utime* argument is NULL, the access and modification times of the file are set to the current time. The effective user ID of the process must match the owner of the file, or the process must have write permission to the file or appropriate privileges, to use the *utime()* function in this manner.

If the *utime* argument is not NULL, it is interpreted as a pointer to a *utimbuf* structure, and the access and modification times are set to the values contained in the designated structure. Only the owner of the file and processes weith appropriate privileges shall be permitted to use the *utime()* function in this way.

The *utimbuf* structure is defined by the header <utime.h> and includes the following members:

Member Type	Member Name	Description
time_t	*actime*	Access time
time_t	*modtime*	Modification time

The times the *utimbuf* structure are measured in seconds since the Epoc.

Upon successful completion, the *utime()* function shall mark for update the *st_ctime* field of the file.

Implementation Considerations

On the HP 3000, file times are updated at file close, not file open as with many other platforms. Therefore, a `utime()` call must follow an explicit `close()` call to change file times. This also means not allowing an implicit file close at the end of execution.

Based on the MPE/iX file system security, this implementation defines that the appropriate privilege which allows the calling process to use the `utime` function to modify time stamps is either the SM (System Manager) capability or AM capability for the specified file, i.e., the calling process' GID matches the file's GID.

An error condition was added to indicate that a file or directory is inaccessible because the ACD associated with it does not have write access, or to designate that a pathname that begins with two slashes was detected. Such pathnames are reserved by this implementation for future consideration.

Errors

If any of the following conditions occur, the *utime()* function will return -1 and set `errno` to the corresponding value.

`EACCES`	CAUSE	One of the following: ■ The calling process does not have search permission to a component of the pathname. ■ The calling process does not have execute permission to the file. ■ The file is not a valid executable file.
	ACTION	One of the following: ■ Make sure that the calling process has search permission to all components of the pathname. ■ Make sure that the calling process has execute permission to the file. ■ Make sure that the file has an MPE/iX file code of `NMPRG`.
`ENAMETOOLONG`	CAUSE	One of the following: ■ The length of the pathname exceeds the `{PATH_MAX}` limit (defined in the file `<limits.h>`). ■ A component of the pathname is longer than `{NAME_MAX}` (defined in `<limits.h>`), and `{_POSIX_NO_TRUNC}` is in effect for that directory.
	ACTION	Make sure that both the component's length and the full pathname length do not exceed the `{NAME_MAX}` or `{PATH_MAX}` limits.
`ENOTDIR`	CAUSE	A component of the pathname is not a directory.
	ACTION	Specify a valid pathname.
`ENOENT`	CAUSE	a component of the pathname for the executable file does not exist, or *pathname* points to an empty string.
	ACTION	Specify a valid pathname.

`EPERM`	CAUSE	One of the following: ■ The calling process does not have the MPE/iX process handling (PH) capability. ■ The calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
	ACTION	One of the following: ■ Link the program file with the MPE/iX PH capability. ■ Do not execute `execl()` when the calling process has outstanding switches to CM code, has set critical mode, has outstanding NOWAITIO, or is holding an operating system internal resource.
`EROFS`	CAUSE	The named file resides on a read-only file system.
	ACTION	Do not attempt to set the times for the file.

See Also

`<sys/stat.h>`, POSIX.1 (Section 5.6.1)

wait

Suspends the calling process to wait for exit status of child processes.

Syntax

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *stat_loc);
```

Parameters

stat_loc A pointer to the exit status of the child process. No information is returned if *stat_loc* is NULL.

Return Values

>0	Success. The process ID of a child process that has terminated is returned.
-1	An error occurred. There is no result, and `errno` is set to indicate the error condition.

Description

The `wait()` function suspends the calling process until status information for one of its terminated child processes is available. If status information is already available, `wait()` returns immediately.

If the calling process receives a signal whose action is to terminate, the calling process terminates. If the calling process receives a signal whose action is to execute a signal handling function, `wait()` returns to the calling process.

If status is available for more than one process, the order in which their status is reported may not correspond to the order of their termination.

The `wait()` function returns to the argument pointed to by *stat_loc* an exit status of 0 if, and only if, the child process that returned status took one of the following two actions:

- returned a value of zero from its `main()` function (outer block)
- passed a status value of zero to `_exit()` or `exit()`

The following macros that evaluate the *stat_loc* parameter, regardless of its value, are defined in the header `<sys/wait.h>`:

`WIFEXITED(exit_status)`	Evaluates to a nonzero value if status was returned for a child process that terminated normally.
`WEXITSTATUS(exit_status)`	If `WIFEXITED` is nonzero, this macro evaluates to the low-order 8 bits of the *stat_loc* parameter that the child process passed to `_exit()` or `exit()`, or the value that the child process returned from `main()`.
`WIFSIGNALED(exit_status)`	Evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught.
`WTERMSIG(exit_status)`	If `WIFSIGNALED` is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.
`WIFSTOPPED(exit_status)`	Evaluates to a nonzero value if status was returned for a child process that is currently stopped.
`WSTOPSIG(exit_status)`	If `WIFSTOPPED` is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.

Implementation Considerations

Refer to the `EFAULT` error description below.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are terminated.

Errors

If an error occurs, `errno` is set to one of the following values:

`ECHILD`	CAUSE	The calling process has no unwaited-for child processes.
	ACTION	No action is necessary.
`EFAULT`	CAUSE	The system detected a bad address in attempting to use the *stat_loc* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINTR`	CAUSE	The function was interrupted by a signal. The *stat_loc* parameter's value is undefined.
	ACTION	Call the `wait()` function again to continue waiting.

See Also

`_exit()`, `fork()`, `pause()`, `waitpid()`, `<signal.h>`, POSIX.1 (Section 3.2.1)

waitpid

Suspends the calling process to wait for exit status of the specified child processes.

Syntax

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

Parameters

pid An parameter of type `pid_t` that specifies the process whose status is being reported. Following are valid values and their meanings:

- >0 A single child process with process ID equal to *pid*.
- 0 All child processes with process group ID equal to the caller's.
- -1 All child processes.
- $<$-1 All child processes with process group ID equal to the absolute value of *pid*.

stat_loc A pointer to the exit status of the child process. No information is stored if *stat_loc* is NULL.

options Modifies the behavior of the `waitpid()` function. The contents of *options* is a bitwise inclusive OR of the following flags (defined in `<sys/wait.h>`):

- `WNOHANG` Do not suspend the calling process when no child status is available.
- `WHANG` Suspend and wait for a child status if none is yet available.
- `WUNTRACED` Suspend and wait for the status of a stopped child process.

Return Values

>0	Success. The process ID of a terminated child process whose process ID matches *pid* is returned.
0	The `WNOHANG` option is specified in *options* and no child specified by *pid* has terminated.
-1	An error occurred. There is no result, and `errno` is set to indicate the error condition.

Description

The `waitpid()` function suspends the calling process until status information for the specified child process(es) is available. If status information is already available, `wait()` returns immediately.

If the calling process receives a signal whose action is to terminate, the calling process terminates. If the calling process receives a signal whose action is to execute a signal handling function, `waitpid()` returns to the calling process.

If status is available for more than one specified process, the order in which their status is reported may not correspond to the order of their termination.

The `waitpid()` function is identical to the `wait()` function when the *pid* parameter has a value of -1 and *options* is equal to zero.

The `waitpid()` function returns to the argument pointed to by *stat_loc* an exit status of 0 if, and only if, the child process that returned status took one of the following two actions:

- returned a value of zero from its `main()` function (outer block)
- passed a status value of zero to `_exit()` or `exit()`

The following macros that evaluate the *stat_loc* parameter, regardless of its value, are defined in the header `<sys/wait.h>`:

`WIFEXITED(exit_status)`	Evaluates to a nonzero value if status was returned for a child process that terminated normally.
`WEXITSTATUS(exit_status)`	If `WIFEXITED` is nonzero, this macro evaluates to the low-order 8 bits of the *stat_loc* parameter that the child process passed to `_exit()` or

	`exit()`, or the value that the child process returned from `main()`.
`WIFSIGNALED(exit_status)`	Evaluates to a nonzero value if status was returned for a child process that terminated due to the receipt of a signal that was not caught.
`WTERMSIG(exit_status)`	If `WIFSIGNALED` is nonzero, this macro evaluates to the number of the signal that caused the termination of the child process.
`WIFSTOPPED(exit_status)`	Evaluates to a nonzero value if status was returned for a child process that is currently stopped.
`WSTOPSIG(exit_status)`	If `WIFSTOPPED` is nonzero, this macro evaluates to the number of the signal that caused the child process to stop.

Implementation Considerations

Refer to the `EFAULT` error description below.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are terminated.

Errors

If an error occurs, `errno` is set to one of the following values:

`ECHILD`	CAUSE	The process or process group specified by *pid* does not exist or is not a child of the calling process.
	ACTION	No action is necessary.
`EFAULT`	CAUSE	The system detected a bad address in attempting to use the *stat_loc* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EINTR`	CAUSE	The function was interrupted by a signal. The *stat_loc* parameter's value is undefined.
	ACTION	Call the `waitpid()` function again to continue waiting.
`EINVAL`	CAUSE	The *options* parameter is invalid.
	ACTION	Specify a valid option as defined in the file `<wait.h>`.

See Also

_exit(), fork(), pause(), wait(), <signal.h>, POSIX.1 (Section 3.2.1)

wordexp

Expand special constructions.

Syntax

```
#include <wordexp.h>
int wordexp (const char *words, wordexp_t *expansions, int flags);
```

Parameters

words is a string containing various special constructions that are typically expanded in shell command lines (for example, command substitutions, directory substitutions, parameter expansions, and so on).

expansions points to an object of type `wordexp_t` where `wordexp()` can store the expanded version of words. It is created by the caller.

flags is a collection of *flags* controlling the `wordexp()` action. Flags are specified by ORing together symbolic constants defined in <wordexp.h>. Possible symbols are:

`WRDE_APPEND`	appends expansions to an existing expansions list generated by a previous call to `wordexp()`.
`WRDE_DOOFFS`	uses the `we_offs` field in the `wordexp_t` structure expansions.
`WRDE_NOCMD`	does not perform command substitution. `wordexp()` fails if command substitution is attempted.
`WRDE_REUSE`	expansions was passed to previous successful call to `wordexp()` and has not been passed to `wordfree()`. The result is the same as if the application had called `wordfree()` and then called `wordexp()`.
`WRDE_SHOWERR`	Do not redirect the standard error to `/dev/null`.

`WRDE_UNDEF`	Reports error on attempt to expand an undefined shell variable.

Return Values

If `wordexp()` completes successfully, it returns zero. Otherwise, it returns one of the following error messages.

Description

`wordexp()` expands special constructs in the string words. See the section Expanded Constructs for a list of the constructs that `wordexp()` expands.

`wordexp()` returns the expansion using a `wordexp_t` structure. This structure has the following fields:

int we_wordc;	is the number of words in the expansion of words. This field is set by `wordexp()`.
char **we_wordv;	points to a list of strings giving the expansions of words from *words*. Each individual field created during expansion becomes a separate word in the `we_wordv` list. The first pointer after the last path name is NULL. This field is set by `wordexp()`.
int we_offs;	tells how many NULL pointers you want at the beginning of the `we_wordv` list. This creates a specified amount of 'blank' space at the beginning of `we_wordv` that can be used for other purposes. For example, you might fill this space with other arguments before passing the whole `we_wordv` vector as an argument to a function like `execv()`.

You set `we_offs` before calling `wordexp()`. `wordexp()` puts the specified number of NULL pointers at the beginning of the `we_wordv` list before putting in pointers to the strings giving the expansion of words. `wordexp()` only pays attention to `we_offs` if `WRDE_DOOFFS` is set in flags.

If `WRDE_APPEND` is specified, `wordexp()` follows these rules:

1. If `WRDE_DOOFFS` is set in the first call to `wordexp()`, it must be set in subsequent calls and `we_offs` must have the same value in each call.

2. If `WRDE_DOOFFS` is not set in the first call, it must not be set in subsequent calls.
3. After the second call, `we_wordv` points to a list containing: the number of NULL pointers as determined by `WRDE_DOOFFS` and `we_offs`; pointers to the words that were in the list before the second call, in the same order as before; and pointers to the new words obtained by the second call, in the order dictated by the flags for the second call.
4. `we_wordc` gives the total number of words from all the calls.

You must not change `we_wordc` or `we_wordv` between calls to `wordexp()`.

Expanded Constructs

`wordexp()` performs the following expansions in the order given:

Directory Substitution
: expands constructs of the form ˜logname into the full path name of the user logname's home directory.

Parameter Expansion
: expands the following constructs:

  ${*parameter*}
  ${*parameter*:-word}
  ${*parameter*:=word}
  ${*parameter*:?word}
  ${*parameter*:+word}
  ${#*parameter*}
  ${*parameter*word}
  ${*parameter*word}
  ${*parameter*#word}
  ${*parameter*##word}

  The result of the expansion of these constructs is detailed in the `sh(1)` man page.

Command Substitution
: expands the following constructs

  $(*command*)
  ‘*command*‘

  by executing *command* and replacing the construct with its output.

Arithmetic Expansion	expands constructs of the form $((%%expression%%)) by replacing the construct with the value of the arithmetic *expression*.

These expansions are discussed in much greater detail in `sh(1)`.

After performing the various expansions, `wordexp()` breaks up the result into separate words. Words are assumed to be separated by any one of the characters in the string value of the environment variable `IFS`. If `IFS` does not exist, `wordexp()` assumes that words are separated by one or more white space characters (blanks, tabs, or newlines). When splitting words in this way, `wordexp()`

After breaking up words into these separate words, `wordexp()` performs filename generation on names that are not quoted (see `sh(1)`).

Finally, `wordexp()` removes backslashes, quotes, and apostrophes from expanded strings, as appropriate.

Note `wordexp()` does not handle the special meanings of `|` (pipe), `&` (put job in background), `;` (separate one command from another), `<` (input redirection) or `>` (output redirection). If the words string contains any of these outside of quotes or apostrophes, `wordexp()` fails and does not expand any words.

Once a program has finished using the paths structure, it should use `wordfree()` to free up the space used to store the path name list.

Errors

If an error occurs, `errno` is set to one of the following values:

`WRDE_BADCHAR`	CAUSE	An unquoted shell metacharacter appeared in words in an inappropriate context.
	ACTION	Examine and correct the syntax of the words string that was passed to the function.
`WRDE_BADVAL`	CAUSE	You attempted to reference an undefined shell variable when `WRDE_UNDEF` was set in *flags*.
	ACTION	Unset `WRDE_UNDEF`, or do not use undefined shell variables.
`WRDE_CMDSUB`	CAUSE	Command substitution was requested when `WRDE_NOCMD` was set in *flags*.
	ACTION	Unset `WRDE_NOCMD`, or do not request command substitution.
`WRDE_ERRNO`	CAUSE	A system call failed inside `wrdexp()`, setting the variable errno. The failure was one of the following: 1. `confstr()` failed to get the name of the shell 2. `pipe()` was unable to create a pipe 3. `wordexp()` was unable to `fork()` or `exec()` the shell. 4. `waitpid()` for a child process failed.
	ACTION	Check the value of the variable `errno` to determine the true cause of the error.
`WRDE_NOSPACE`	CAUSE	`wordexp()` was unable to allocate memory for one of its operations. `we_wordc` and `we_wordv` are still updated to show whatever words have already expanded.
	ACTION	Free up more memory.
`WRDE_SYNTAX`	CAUSE	words contained a shell syntax error, such as an unbalanced parentheses or unterminated string.
	ACTION	Correct the syntax of words.

See Also

sh(1), glob(), wordfree()

wordfree

Release data created by wordexp.

Syntax

```
#include <wordexp.h>
void wordfree(wordexp_t *expansions);
```

Parameters

expansions	Is a `wordexp_t` structure used in a previous call to `wordexp()`.

Description

`wordfree()` frees any memory allocated in connection with the the *expansions* structure. Typically, this gets rid of any space that a call to `wordexp()` allocated to hold a word expansion list.

See Also

`sh(1)`, `wordexp()`

write

Writes data to a file.

Syntax

```
ssize_t write (int fildes, const void *buffer, size_t nbyte);
```

Parameters

fildes	An open file descriptor.
buffer	A pointer to a buffer containing data to be written. The size of the buffer must be greater than *nbyte*.
nbyte	The maximum number of bytes to write.

Return Values

>=0	Success. An integer indicating the number of bytes actually written is returned.
-1	An error occurred. No data is written, and `errno` is set to indicate the error condition.

Description

The `write()` function attempts to writes *nbyte* bytes from the buffer pointed to by *buffer* to the open file associated with the open file descriptor *fildes*.

On a file capable of seeking, `write()` starts from the current file offset position. Before successful return from `write()`, the file offset is incremented by the number of bytes actually written. If the incremented file offset is greater than the EOF of the file, the EOF of the file is set to the new file offset.

If the `O_APPEND` file status flag is set, the file offset is set to the end of the file prior to each write.

On a file not capable of seeking, `write()` starts from the current position. (The file offset for such a file is undefined.)

Upon successful completion, the `write()` function returns the actual number of bytes written to the file and, if *nbyte* is greater than 0, marks for update the `st_ctime` and `st_mtime` time fields of the file.

If `write()` requests that more bytes be written than there is room for (for example, the physical end of medium), only as many bytes as there is room for are written. In this case, the next write to the file of a nonzero number of bytes fails, and an error is returned.

If *nbytes* is zero, the `write()` function writes zero bytes of data. In this case, the file offset position is not changed and no time fields are marked for update.

Implementation Considerations

Refer to the `EFBIG`, `EFAULT`, `EIMPL`, and `ESYSERR` error descriptions below.

Signals generated for the calling process during execution of `write()` are deferred from delivery until completion of this function.

Errors

If an error occurs, `errno` is set to one of the following values:

`EBADF`	CAUSE	The *fildes* parameter is not a valid open file descriptor open for writing.
	ACTION	Pass a valid open file descriptor of a file open for writing by the calling process.
`EFAULT`	CAUSE	The system detected a NULL or bad address in attempting to use the *buffer* parameter.
	ACTION	Make sure that the pointer is correctly initialized.
`EFBIG`	CAUSE	The file size has exceeded the file limit. The default file limit is 2 gigabytes.
	ACTION	Reduce the size of the file.

`EIMPL`	CAUSE	The file size has exceeded the disk space limit established by the MPE/iX accounting facility.
	ACTION	Make sure that the MPE/iX accounting facility allows you to increase the size of the file.
`ENOSPC`	CAUSE	There is no free space remaining on the device containing the file.
	ACTION	Deallocate space on the device.
`ESYSERR`	CAUSE	An operating system error has occurred that does not map directly to any of the above errors.
	ACTION	Examine the MPE/iX error stack for the type of system error.

See Also

`creat()`, `dup()`, `lseek()`, `open()`, POSIX.1 (Section 6.4.2)

5

POSIX/iX Header Descriptions

This chapter describes the contents of the header files provided with the POSIX/iX library. The POSIX.1 extensions are invoked by the `_POSIX_SOURCE` feature test macro.

Note The `_POSIX_SOURCE` feature test macro must be specified in your source code before you include any headers described in this chapter.

The header or headers required for each function are specified in the syntax descriptions provided in this manual and in the *HP C/iX Library Reference Manual* (30026-90001).

To reference a POSIX/iX library header, place the `#include` preprocessor directive in your source code. The order of inclusion of the header files may be significant. Include the header in the order described in each POSIX/iX library function description.

The syntax for including a header file is:

```
#include <headername.h>
```

By enclosing *headername* in angle brackets (< >), you instruct the compiler to look for that header in `/usr/include`.

For example, if you want to use the `open()` function, your program must specify three headers:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Header file identifiers beginning with an underscore (_) are reserved for library use. You should not create identifiers that begin with an underscore within your source code.

The following headers are not described in this manual. Like all the headers provided with the MPE/iX Developer's Kit, they are located under the directory `/usr/include`. You can view them online from the MPE/iX CI using the `PRINT` command or from the MPE/iX Shell using the `cat` command.

- `glob.h`
- `regex.h`
- `wordexp.h`

The following table lists each of the POSIX/iX library headers and a brief description of each header. Remaining sections of this chapter describe the contents of the headers not already described in the *HP C/iX Library Reference Manual* (30026-90001).

Table 5-1. POSIX/iX Library Headers

Header	Description	Description Location
`<assert.h>`	Defines the `assert()` macro.	*HP C/iX Library Reference Manual*
`<ctype.h>`	Declares macros and functions useful for testing and mapping characters.	*HP C/iX Library Reference Manual*
`<dirent.h>`	Declares functions and data structures used for managing directories.	This chapter
`<errno.h>`	Declares error variables and defines macros useful for obtaining a more detailed description of a library function error.	This chapter
`<fcntl.h>`	Defines the `creat()`, `fcntl()`, and `open()` functions as well as macros used by these functions.	This chapter
`<float.h>`	Defines macros that describe the floating-point types.	*HP C/iX Library Reference Manual*
`<limits.h>`	Defines implementation limits for POSIX/iX.	This chapter
`<locale.h>`	Used for localization. Contains macro definitions, function, and type declarations needed to select the desired locale.	*HP C/iX Library Reference Manual*
`<malloc.h>`	Declares memory management functions, `mallopt()` argument functions, and a structure returned by the `mallinfo()` function. Memory management functions are also declared in `<stdlib.h>`.	*HP C/iX Library Reference Manual*
`<math.h>`	Contains declarations for the POSIX/iX math library functions, as well as functions in the standard library that return floating-point values. Also defines the structure and constants used by the `matherr` error-handling mechanisms.	*HP C/iX Library Reference Manual*
`<memory.h>`	Declares several functions useful for manipulating character arrays and other objects treated as character arrays. These functions are also declared in `<string.h>`.	*HP C/iX Library Reference Manual*
`<mpe.h>`	Declares several types, constants and functions that facilitate MPE operating system interface.	*HP C/iX Library Reference Manual*
`<search.h>`	Defines the types used with the `hsearch()` and `tsearch()` functions.	*HP C/iX Library Reference Manual*

Table 5-1. POSIX/iX Library Headers (continued)

Header	Description	Description Location
`<setjmp.h>`	Declares a type and several functions for bypassing the normal function call and return discipline.	This chapter
`<signal.h>`	Contains declaration used in dealing with conditions that may be reported during program execution.	This chapter
`<stdarg.h>`	Provides a standard method for dealing with variable arguments.	*HP C/iX Library Reference Manual*
`<stddef.h>`	Defines several macros and types required by ANSI C functions.	*HP C/iX Library Reference Manual*
`<stdio.h>`	Defines a structure and several functions and macros useful for I/O.	This chapter
`<stdlib.h>`	Declares various ANSI C general utility functions and macros.	*HP C/iX Library Reference Manual*
`<string.h>`	Declares functions useful for manipulating character arrays and other objects treated as character arrays.	*HP C/iX Library Reference Manual*
`<sys/stat.h>`	Declares the **chmod()**, **fstat()**, **mkdir()**, **stat()**, and **umask()** functions and their required data types and symbols.	This chapter
`<sys/times.h>`	Contains the definition of the struct **tms**.	*HP C/iX Library Reference Manual*
`<sys/types.h>`	Defines fundamental types required by POSIX.1 conforming functions.	This chapter
`<sys/wait.h>`	Declares the **wait()** and **waitpid()** functions.	This chapter
`<time.h>`	Declares types, global variables, and functions used for manipulating time.	This chapter
`<unistd.h>`	Defines various miscellaneous POSIX.1 conforming macros and functions. Some of these macros are also declared in `<stdarg.h>`.	This chapter
`<values.h>`	Contains a set of manifest constants, conditionally defined for particular processor architectures.	*HP C/iX Library Reference Manual*
`<varargs.h>`	Declares types and macros for declaring variable argument functions. See also `<stdarg.h>`.	*HP C/iX Library Reference Manual*

Index

B

C

D

E

F

G

M

N

O

P

R

S

T

U

V

W