Technical Standard

Extended API Set Part 2

The Open Group

Technical Standard

Extended API Set Part 2

ISBN: 1-931624-67-4
Document Number: C063

Published in the U.K. by The Open Group, October 2006.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Thames Tower
37-45 Station Road
Reading
Berkshire, RG1 1LX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX certification.

Further information on The Open Group is available at *www.opengroup.org*.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at *www.opengroup.org/certification*.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at *www.opengroup.org/bookstore*.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published at *www.opengroup.org/corrigenda*.

This Document

This document has been prepared by The Open Group Base Working Group. The Open Group Base Working Group is considering submitting a number of API sets to the Austin Group as input to the revision of the Base Specifications, Issue 6.

This is the second document in that set.

Trademarks

Acknowledgements

The contributions of the following to the development of this document are gratefully acknowledged:

- The Open Group Base Working Group

Chapter 1

Introduction

1.1 Scope

The purpose of this document is to define a set of new API extensions to further increase application capture and hence portability for systems built upon the Single UNIX Specification, Version 3.

The scope of this set of extensions has been to consider a set of interfaces drawn from existing implementations that address a number of common problems with existing functions.

1.2 Relationship to Other Formal Standards

No decision has been made on whether these interfaces will be added to a future Technical Standard of The Open Group, how these interfaces would announce themselves in the name space, or whether related interfaces should be merged with existing reference pages. This Technical Standard is being forwarded to the Austin Group for consideration as input to the revision of the Base Specifications, Issue 6.

Chapter 2

Changes to the Base Definitions Volume

It is proposed that these additions comprise a new Option Group called the Extended Interfaces.

2.1 Section 1.5.1, Codes

Add a new margin code as follows:

UX Extended Interfaces

The functionality described is optional. The functionality described is also an extension to the ISO C standard.

Where applicable, functions are marked with the UX margin legend in the SYNOPSIS section. Where additional semantics apply to a function, the material is identified by use of the UX margin legend.

Notes:

1. This section is repeated in XBD, XSH, and XCU and therefore will appear in XBD (Section 1.5.1), XSH (Section 1.8.1), and XCU (Section 1.8.1).
2. The use of UX as a margin code is a placeholder and may change in the final publication.

2.2 Chapter 13, Headers

The following header file reference pages will need the following additions or changes. The additions should be marked with the UX margin legend and shaded as part of the Extended Interfaces Option Group.

<dirent.h>

UX The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
DIR *fdopendir(int);
```

<fcntl.h>

In the Base Definitions volume of IEEE Std 1003.1-2001, Page 224, change Lines 7859-7862 from:

File access modes used for *open*( ) and *fcntl*( ) are as follows:

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

to:

File access modes used for *open*( ) and *fcntl*( ) are as follows:

UX	O_EXEC	Open for execute only (non-directory files). Use of this flag on directories is currently unspecified.

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

Add a statement in FUTURE DIRECTIONS on Page 225, Line 7903:

The meaning of the O_EXEC flag on directories may be specified in a future version.

UX The following value is a special value used in place of a file descriptor:

AT_FDCWD	Use the current working directory to determine the target of relative file paths.

The following is a value for *flag* used by *faccessat*( ):

AT_EACCESS	Check access using effective user and group ID.

The following is a value for *flag* used by *fstatat*( ), *fchmodat*( ), and *fchownat*( ):

AT_SYMLINK_NOFOLLOW	Do not follow symbolic links.

The following is a value for *flag* used by *linkat*( ):

AT_SYMLINK_FOLLOW	Follow symbolic link.

The following is a value for *flag* used by *open*( ) and *openat*( ):

O_DIRECTORY	Fail if not a directory.
O_NOFOLLOW	Do not follow symbolic links.

The following is a value for *flag* used by *unlinkat*( ):

AT_REMOVEDIR	Remove directory instead of file.

The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int openat(int, const char *, int, ...);
```

<stdio.h>

UX The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int renameat(int, const char *, int, const char *);
```

<sys/socket.h>

Add to the DESCRIPTION section of **<sys/socket.h>** after MSG_OOB on Line 12593:

UX MSG_NOSIGNAL	No SIGPIPE generated when an attempt to send is made on a stream-oriented socket that is no longer connected.

<sys/stat.h>

UX The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int fstatat(int, const char *, struct stat *, int);
int mkdirat(int, const char *, mode_t);
int mkfifoat(int, const char *, mode_t);
int mknodat(int, const char *, mode_t, dev_t);
```

<sys/time.h>

UX The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int futimesat(int, const char *, const struct timeval [2]);
```

<unistd.h>

UX The following shall be declared as functions and may also be defined as macros. Function prototypes shall be provided.

```
int faccessat(int, const char *, int);
int fchmodat(int, const char *, mode_t, int);
int fchownat(int, const char *, uid_t, gid_t, int);
int fexecve(int, char *const [], char *const []);
int linkat(int, const char *, int, const char *, int flag);
ssizt_t readlinkat(int, const char *, char *, size_t);
int symlinkat(const char *, int, const char *);
int unlinkat(int, const char *, int);
```

Chapter 3

Changes to the System Interfaces Volume

3.1 Changes to Sockets-Related Reference Pages

Add the following to the text describing the *flags* argument after MSG_OOB within the DESCRIPTION section of *send*( ), *sendmsg*( ), and *sendto*( ):

MSG_NOSIGNAL	Requests not to send the SIGPIPE signal if an attempt to send is made on a stream-oriented socket that is no longer connected. The [EPIPE] error shall still be returned.

3.2 Changes to File-Related Reference Pages

Make the following changes to the *open*( ) reference page (the System Interfaces volume of IEEE Std 1003.1-2001, Page 850).

Change from:

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**. Applications shall specify exactly one of the first three values (file access modes) below in the value of *oflag*:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

to:

Values for *oflag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**. Applications shall specify exactly one of the first four values (file access modes) below in the value of *oflag*:

UX	O_EXEC	Open for execute only (non-directory files). Use of this flag on directories is currently unspecified.
	O_RDONLY	Open for reading only.
	O_WRONLY	Open for writing only.
	O_RDWR	Open for reading and writing. The result is undefined if this flag is applied to a FIFO.

Add the following description between O_CREAT and O_DSYNC:

O_DIRECTORY	If *path* does not name a directory, fail and set *errno* to [ENOTDIR].

Add the following description between O_NOCTTY and O_NONBLOCK:

O_NOFOLLOW	If *path* names a symbolic link, fail and set *errno* to [ELOOP].

Under O_TRUNC, change from:

The result of using O_TRUNC with O_RDONLY is undefined.

to:

The result of using O_TRUNC without either O_RDWR or O_WRONLY is undefined.

In the ERRORS section, change from:

[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument.

to:

[ELOOP]	A loop exists in symbolic links encountered during resolution of the *path* argument, or O_NOFOLLOW was specified and the *path* argument names a symbolic link.

and change from:

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

to:

[ENOTDIR]	A component of the path prefix is not a directory, or O_DIRECTORY was specified and the *path* argument does not name a directory.

In the RATIONALE, insert the following text after the paragraphs talking about symbolic links O_CREAT and O_EXCL (System Interfaces volume of IEEE Std 1003.1-2001, Page 855, Line 27922):

In addition, the *open*( ) function refuses to open non-directories if the O_DIRECTORY flag is set. This avoids race conditions whereby a user might compromise the system by substituting a hard link to a sensitive file (e.g., a device or a FIFO) while a privileged application is running, where opening a file even for read access might have undesirable side-effects.

In addition, the *open*( ) function does not follow symbolic links if the O_NOFOLLOW flag is set. This avoids race conditions whereby a user might compromise the system by substituting a symbolic link to a sensitive file (e.g., a device) while a privileged application is running, where opening a file even for read access might have undesirable side-effects.

Add rationale for *open*( ). On Page 855, after Line 27916 (after O_RDONLY | O_WRONLY == O_RDWR), insert the following text:

O_EXEC is specified as one of the four file access modes. On implementations where none of O_RDONLY, O_WRONLY, or O_RDWR is zero, applications may open a directory with O_EXEC OR'd in with one of the other three file access modes. On many historical implementations, this cannot be done since O_RDONLY has been defined to be zero.

Add a statement to FUTURE DIRECTIONS as follows:

The meaning of the O_EXEC flag on directories may be specified in a future version.

Make the following change to the *opendir*( ) reference page.

At the end of the DESCRIPTION, add:

If the type **DIR** is implemented using a file descriptor, the descriptor shall be obtained as if the O_DIRECTORY flag was passed to *open*( ).

3.3 Reference Pages

Add the following new system interface descriptions in alphabetical order with the existing system interface descriptions in Chapter 3, System Interfaces.

NAME

faccessat — determine accessibility of a file relative to directory file descriptor

SYNOPSIS

UX
```
#include <unistd.h>

int faccessat(int fd, const char *path, int amode, int flag);
```

DESCRIPTION

The *faccessat*( ) function shall be equivalent to the *access*( ) function except in the case where *path* specifies a relative path. In this case the file whose accessibility is to be determined shall be located relative to the directory associated with the file descriptor *fd* instead of the current working directory.

If *faccessat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *access*( ).

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_EACCESS	The checks for accessibility are performed using the effective user and group IDs instead of the real user and group ID as required in a call to *access*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *access*( ). In addition, the *faccessat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor.

The *faccessat*( ) function may fail if:

[EINVAL]	The value of the *flag* argument is not valid.
[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *faccessat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

The use of the AT_EACCESS value for *flag* enables functionality not available in *access*( ).

RATIONALE

The purpose of the *faccessat*( ) interface is to enable the checking of the accessibility of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *access*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *faccessat*( ) function it can be guaranteed that the file tested for accessibility is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

access( ), *chmod*( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

NAME

fchmodat — change mode of a file relative to directory file descriptor

SYNOPSIS

UX
```
#include <sys/stat.h>

int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

DESCRIPTION

The *fchmodat*( ) function shall be equivalent to the *chmod*( ) function except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, then the mode of the symbolic link is changed.

If *fchmodat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used. If also *flag* is zero, the behavior shall be identical to a call to *chmod*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

Refer to *chmod*( ). In addition, the *fchmodat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *fchmodat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

[EOPNOTSUPP] The AT_SYMLINK_NOFOLLOW bit is set in the *flag* argument, *path* names a symbolic link, and the system does not support changing the mode of a symbolic link.

EXAMPLES

None.

APPLICATION USAGE

The *fchmodat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *fchmodat*( ) interface is to enable changing the mode of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *chmod*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *fchmodat*( ) function it can be guaranteed that the changed file is located relative to the desired directory. Some implementations might allow changing the mode of symbolic links. This is not supported by the interfaces in the POSIX specification. Systems with such support provide an interface named *lchmod*( ). To support such implementations *fchmodat*( ) has a *flag* parameter.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue X.

NAME

fchownat — change owner and group of a file relative to directory file descriptor

SYNOPSIS

UX

```
#include <unistd.h>

int fchownat(int fd, const char *path, uid_t owner, gid_t group,
    int flag);
```

DESCRIPTION

The *fchownat*( ) function shall be equivalent to the *chown*( ) and *lchown*( ) functions except in the case where *path* specifies a relative path. In this case the file to be changed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, ownership of the symbolic link is changed.

If *fchownat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *chown*( ) or *lchown*( ) respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in the *flag* argument.

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *lchown*( ). In addition, the *fchownat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *fchownat*( ) function may fail if:

[EINVAL]	The value of the *flag* argument is not valid.
[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.
[EOPNOTSUPP]	The *path* argument names a symbolic link and the implementation does not support setting the owner or group of a symbolic link.

EXAMPLES

None.

APPLICATION USAGE

The *fchownat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *fchownat*( ) interface is to enable changing ownership of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *chown*( ) or *lchown*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *fchownat*( ) function it can be guaranteed that the changed file is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

chown( ), *lchown*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

NAME

fdopendir — open directory associated with file descriptor

SYNOPSIS

UX
```
#include <dirent.h>

DIR *fdopendir(int fd);
```

DESCRIPTION

The *fdopendir*( ) function shall be equivalent to the *opendir*( ) function except that the directory is specified by a file descriptor rather than by a name. The file offset associated with the file descriptor at the time of the call determines which entries are returned.

Upon successful return from *fdopendir*( ), the file descriptor is under the control of the system, and if any attempt is made to close the file descriptor, or to modify the state of the associated description other than by means of *closedir*( ), *readdir*( ), *readdir_r*( ), or *rewinddir*( ), the behavior is implementation-defined. Upon calling *closedir*( ) the file descriptor shall be closed.

It is unspecified whether the FD_CLOEXEC flag will be set on the file descriptor by a successful call to *fdopendir*( ).

RETURN VALUE

Upon successful completion, *fdopendir*( ) shall return a pointer to an object of type **DIR**. Otherwise, a null pointer shall be returned and *errno* set to indicate the error.

ERRORS

The *fdopendir*( ) function shall fail if:

[EBADF]	The *fd* argument is not a valid file descriptor open for searching.
[ENOTDIR]	The descriptor *fd* is not associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *fdopendir*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *fdopendir*( ) interface is to enable opening files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *opendir*( ), resulting in unspecified behavior.

FUTURE DIRECTIONS

None.

SEE ALSO

closedir( ), *open*( ), *openat*( ), *opendir*( ), *readdir*( ), *readdir_r*( ), *rewinddir*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<dirent.h>**

CHANGE HISTORY

First released in Issue X.

NAME

fexecve — execute a file

SYNOPSIS

UX
```
#include <unistd.h>

int fexecve(int fd, char *const argv[], char *const envp[]);
```

DESCRIPTION

The *fexecve*( ) function shall be equivalent to the *execve*( ) function except that the file to be executed is determined by the file descriptor *fd* instead of a pathname.

The file offset of *fd* is ignored.

RETURN VALUE

If the *fexecve*( ) function returns to the calling process image, an error has occurred; the return value shall be −1, and *errno* shall be set to indicate the error.

ERRORS

The *fexecve*( ) function shall fail if:

[E2BIG]	The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.
[EACCESS]	The new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.
[EBADF]	The *fd* argument is not a valid file descriptor open for executing.
[EINVAL]	The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.
[ENOEXEC]	The new process image file has the appropriate access permission but has an unrecognized format.

The *fexecve*( ) function may fail if:

[ENOMEM]	The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.
[ETXTBSY]	The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

EXAMPLES

None.

APPLICATION USAGE

The *fexecve*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

If an application wants to perform a checksum test of the file being executed before executing it, the file will need to be opened with read permission to perform the checksum test.

Since execute permission is checked by *fexecve*( ), the file description *fd* need not have been opened with the O_EXEC flag. However, if the file to be executed denies read and write permission for the process preparing to do the *exec*, the only way to provide the *fd* to *fexecve*( ) will be to use the O_EXEC flag when opening *fd*. In this case, the application will not be able to perform a checksum test since it will not be able to read the contents of the file.

Note that when a file descriptor is opened with O_RDONLY, O_RDWR, or O_WRONLY mode, the file descriptor can be used to read, read and write, or write the file, respectively, even if the mode of the file changes after the file was opened. Using the O_EXEC open mode is different; *fexecve*( ) will ignore the mode that was used when the file descriptor was opened and the *exec* will fail if the mode of the file associated with *fd* does not grant execute permission to the calling process at the time *fexecve*( ) is called.

RATIONALE

The purpose of the *fexecve*( ) interface is to enable executing a file which has been verified to be the intended file. It is possible to actively check the file by reading from the file descriptor and be sure that the file is not exchanged for another between the reading and the execution. Alternatively, an interface like *openat*( ) can be used to open a file which has been found by reading the content of a directory using *readdir*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

exec, *open*( ), *openat*( ), *readdir*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

NAME

fstatat — get status of a file relative to directory file descriptor

SYNOPSIS

UX
```
#include <sys/stat.h>

int fstatat(int fd, const char *restrict path,
    struct stat *restrict buf, int flag);
```

DESCRIPTION

The *fstatat*( ) function shall be equivalent to the *stat*( ) and *lstat*( ) functions except in the case where *path* specifies a relative path. In this case the status shall be retrieved from a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_NOFOLLOW
If *path* names a symbolic link, the status of the symbolic link is returned.

If *fstatat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *stat*( ) or *lstat*( ) respectively, depending on whether or not the AT_SYMLINK_NOFOLLOW bit is set in *flag*.

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *lstat*( ). In addition, the *fstatat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *fstatat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *fstatat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *fstatat*( ) interface is to obtain the status of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *stat*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *fstatat*( ) function it can be guaranteed that the file for which status is returned is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

lstat( ), *stat*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue X.

NAME

futimesat — set access and modification time of a file relative to directory file descriptor

SYNOPSIS

UX

```
#include <sys/time.h>

int futimesat(int fd, const char *path, const struct timeval times[2]);
```

DESCRIPTION

The *futimesat*( ) function shall be equivalent to the *utimes*( ) function except in the case where *path* specifies a relative path. In this case the access and modification time is set to that of a file relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

If *futimesat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *utimes*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *utimes*( ). In addition, the *futimesat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *futimesat*( ) function may fail if:

[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *futimesat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *futimesat*( ) interface is to set the access and modification time of files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *utimes*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *futimesat*( ) function it can be guaranteed that the changed file is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

utimes( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/time.h>**

CHANGE HISTORY

First released in Issue X.

NAME

linkat — link one file to another file relative to two directory file descriptors

SYNOPSIS

UX
```
#include <unistd.h>

int linkat(int fd1, const char *path1, int fd2, const char *path2,
    int flag);
```

DESCRIPTION

The *linkat*( ) function shall be equivalent to the *link*( ) function except in the case where either *path1* or *path2* or both are relative paths. In this case a relative path *path1* is interpreted relative to the directory associated with the file descriptor *fd1* instead of the current working directory and similarly for *path2* and the file descriptor *fd2*. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_SYMLINK_FOLLOW
: If *path1* names a symbolic link, a new link for the target of the symbolic link is created.

If *linkat*( ) is passed the special value AT_FDCWD in the *fd1* or *fd2* parameter, the current working directory is used for the respective *path* argument. If both *fd1* and *fd2* have value AT_FDCWD, the behavior shall be identical to a call to *link*( ).

Unless *flag* contains the AT_SYMLINK_FOLLOW flag, if *path1* names a symbolic link, a new link is created for the symbolic link *path1* and not its target.

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return –1 and set *errno* to indicate the error.

ERRORS

Refer to *link*( ). In addition, the *linkat*( ) function shall fail if:

[EBADF] The *path1* or *path2* argument does not specify an absolute path and the *fd1* or *fd2* argument, respectively, is neither AT_FDCWD nor a valid file descriptor open for searching.

The *linkat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path1* or *path2* argument is not an absolute path and *fd1* or *fd2*, respectively, is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *linkat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *linkat*( ) interface is to link files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *link*( ), resulting in unspecified behavior. By opening a file descriptor for the directory of both the existing file and the target location and using the *linkat*( ) function it can be guaranteed that the both filenames are in the desired directories.

The AT_SYMLINK_FOLLOW flag allows for implementing both common behaviors of the *link*( ) function. The POSIX specification requires that if *path1* is a symbolic link, a new link for the target of the symbolic link is created. Many systems by default or as an alternative provide a mechanism to avoid the implicit symlink lookup and create a new link for the symbolic link itself.

FUTURE DIRECTIONS

None.

SEE ALSO

link( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

NAME

mkdirat — make a directory relative to directory file descriptor

SYNOPSIS

UX
```
#include <sys/stat.h>

int mkdirat(int fd, const char *path, mode_t mode);
```

DESCRIPTION

The *mkdirat*( ) function shall be equivalent to the *mkdir*( ) function except in the case where *path* specifies a relative path. In this case the newly created directory is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

If *mkdirat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *mkdir*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *mkdir*( ). In addition, the *mkdirat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *mkdirat*( ) function may fail if:

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *mkdirat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *mkdirat*( ) interface is to create a directory in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to the call to *mkdir*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *mkdirat*( ) function it can be guaranteed that the newly created directory is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

mkdir( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue X.

NAME

mkfifoat — make a FIFO special file relative to directory file descriptor

SYNOPSIS

UX
```
#include <sys/stat.h>

int mkfifoat(int fd, const char *path, mode_t mode);
```

DESCRIPTION

The *mkfifoat*( ) function shall be equivalent to the *mkfifo*( ) function except in the case where *path* specifies a relative path. In this case the newly created FIFO is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

If *mkfifoat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *mkfifo*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *mkfifo*( ). In addition, the *mkfifoat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *mkfifoat*( ) function may fail if:

[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *mkfifoat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *mkfifoat*( ) interface is to create a FIFO special file in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *mkfifo*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *mkfifoat*( ) function it can be guaranteed that the newly created FIFO is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

mkfifo( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue X.

NAME

mknodat — make a directory, a special file, or a regular file relative to directory file descriptor

SYNOPSIS

UX
```
#include <sys/stat.h>

int mknodat(int fd, const char *path, mode_t mode, dev_t dev);
```

DESCRIPTION

The *mknodat*( ) function shall be equivalent to the *mknod*( ) function except in the case where *path* specifies a relative path. In this case the newly created directory, special file, or regular file is located relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

If *mknodat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *mknod*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *mknod*( ). In addition, the *mknodat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *mknodat*( ) function may fail if:

[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *mknodat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *mknodat*( ) interface is to create directories, special files, or regular files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *mknod*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *mknodat*( ) function it can be guaranteed that the newly created directory, special file, or regular file is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

mknod( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<sys/stat.h>**

CHANGE HISTORY

First released in Issue X.

NAME

openat — open file relative to directory file descriptor

SYNOPSIS

UX
```
#include <fcntl.h>

int openat(int fd, const char *path, int flag, ...);
```

DESCRIPTION

The *openat*( ) function shall be equivalent to the *open*( ) function except in the case where *path* specifies a relative path. In this case the file to be opened is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches. The *flag* parameter and the optional fourth parameter correspond exactly to the parameters of *open*( ).

If *openat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *open*( ).

RETURN VALUE

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, –1 shall be returned and *errno* shall be set to indicate the error. No files shall be created or modified if the function returns –1.

ERRORS

Refer to *open*( ). In addition, the *openat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *openat*( ) function may fail if:

[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *openat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *openat*( ) interface is to enable opening files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *open*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *openat*( ) function it can be guaranteed that the opened file is located relative to the desired directory. Some implementations use the *openat*( ) interface for other purposes as well. In some cases, if the *flag* parameter has the O_XATTR bit set, the returned file descriptor provides access to extended attributes. This functionality is not standardized here.

FUTURE DIRECTIONS

None.

SEE ALSO

open( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**

CHANGE HISTORY

First released in Issue X.

NAME

readlinkat — read the content of a symlink relative to a directory file descriptor

SYNOPSIS

XSI
```
#include <unistd.h>

ssize_t readlinkat(int fd, const char *path, char *buf,
    size_t bufsize);
```

DESCRIPTION

The *readlinkat*( ) function shall be equivalent to the *readlink*( ) function except in the case where *path* specifies a relative path. In this case the symbolic link whose content is read is relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

If *readlinkat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *readlink*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *readlink*( ). In addition, the *readlinkat*( ) function shall fail if:

[EBADF]	The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *readlinkat*( ) function may fail if:

[ENOTDIR]	The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *readlinkat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *readlinkat*( ) interface is to read the content of symbolic links in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *readlink*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *readlinkat*( ) function it can be guaranteed that the symbolic link read is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

readlink( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

NAME

renameat — rename a file relative to directory file descriptor

SYNOPSIS

UX
```
#include <stdio.h>

int renameat(int oldfd, const char *old, int newfd,
    const char *new);
```

DESCRIPTION

The *renameat*( ) function shall be equivalent to the *rename*( ) function except in the case where either *old* or *new* specifies a relative path. If *old* is a relative path, the file to be renamed is located relative to the directory associated with the file descriptor *oldfd* instead of the current working directory. If *new* is a relative path, the same happens only relative to the directory associated with *newfd*. The test for whether *oldfd* is searchable is based on whether *oldfd* is open for searching, not whether the underlying directory currently permits searches.

If *renameat*( ) is passed the special value AT_FDCWD in the *oldfd* or *newfd* parameter, the current working directory shall be used in the determination of the file for the respective *path* parameter.

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *rename*( ). In addition, the *renameat*( ) function shall fail if:

[EBADF] The *old* argument does not specify an absolute path and the *oldfd* argument is neither AT_FDCWD nor a valid file descriptor open for searching, or the *new* argument does not specify an absolute path and the *newfd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *renameat*( ) function may fail if:

[ENOTDIR] The *old* argument is not an absolute path and *oldfd* is neither AT_FDCWD nor a file descriptor associated with a directory, or the *new* argument is not an absolute path and *newfd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *renameat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *renameat*( ) interface is to rename files in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *rename*( ), resulting in unspecified behavior. By opening file descriptors for the source and target directories and using the *renameat*( ) function it can be guaranteed that that renamed file is located correctly and the resulting file is in the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

rename( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<stdio.h>**

CHANGE HISTORY

First released in Issue X.

NAME

symlinkat — make a symlink relative to directory file descriptor

SYNOPSIS

UX
```
#include <unistd.h>

int symlinkat(const char *path1, int fd, const char *path2);
```

DESCRIPTION

The *symlinkat*( ) function shall be equivalent to the *symlink*( ) function except in the case where *path2* specifies a relative path. In this case the symbolic link is created relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

If *symlinkat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *symlink*( ).

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *symlink*( ). In addition, the *symlinkat*( ) function shall fail if:

[EBADF]	The *path2* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

The *symlinkat*( ) function may fail if:

[ENOTDIR]	The *path2* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *symlinkat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *symlinkat*( ) interface is to create symbolic links in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *symlink*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *symlinkat*( ) function it can be guaranteed that the created symbolic link is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

symlink( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

NAME

unlinkat — remove a directory entry relative to directory file descriptor

SYNOPSIS

UX
```
#include <unistd.h>

int unlinkat(int fd, const char *path, int flag);
```

DESCRIPTION

The *unlinkat*( ) function shall be equivalent to the *unlink*( ) or *rmdir*( ) function except in the case where *path* specifies a relative path. In this case the directory entry to be removed is determined relative to the directory associated with the file descriptor *fd* instead of the current working directory. The test for whether *fd* is searchable is based on whether *fd* is open for searching, not whether the underlying directory currently permits searches.

Values for *flag* are constructed by a bitwise-inclusive OR of flags from the following list, defined in **<fcntl.h>**:

AT_REMOVEDIR Remove the directory entry specified by *fd* and *path* as a directory, not a normal file.

If *unlinkat*( ) is passed the special value AT_FDCWD in the *fd* parameter, the current working directory is used and the behavior shall be identical to a call to *unlink*( ) or *rmdir*( ) respectively, depending on whether or not the AT_REMOVEDIR bit is set in *flag*.

RETURN VALUE

Upon successful completion, the function shall return 0. Otherwise, it shall return −1 and set *errno* to indicate the error.

ERRORS

Refer to *unlink*( ). In addition, the *unlinkat*( ) function shall fail if:

[EBADF] The *path* argument does not specify an absolute path and the *fd* argument is neither AT_FDCWD nor a valid file descriptor open for searching.

[EEXIST] or [ENOTEMPTY]
The *flag* parameter has the AT_REMOVEDIR bit set and the *path* argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dot-dot.

[ENOTDIR] The *flag* parameter has the AT_REMOVEDIR bit set and *path* does not name a directory.

The *unlinkat*( ) function may fail if:

[EINVAL] The value of the *flag* argument is not valid.

[ENOTDIR] The *path* argument is not an absolute path and *fd* is neither AT_FDCWD nor a file descriptor associated with a directory.

EXAMPLES

None.

APPLICATION USAGE

The *unlinkat*( ) function is part of the Extended Interfaces Option Group and need not be available on all implementations.

RATIONALE

The purpose of the *unlinkat*( ) interface is to remove directory entries in directories other than the current working directory without exposure to race conditions. Any part of the path of a file could be changed in parallel to a call to *unlink*( ), resulting in unspecified behavior. By opening a file descriptor for the target directory and using the *unlinkat*( ) function it can be guaranteed that the removed directory entry is located relative to the desired directory.

FUTURE DIRECTIONS

None.

SEE ALSO

rmdir( ), *unlink*( ), the Base Definitions volume of IEEE Std 1003.1-2001, **<fcntl.h>**, **<unistd.h>**

CHANGE HISTORY

First released in Issue X.

Index