_cpio.h>

NAME

cpio.h − cpio archive values

SYNOPSIS

#include <cpio.h>

DESCRIPTION

Values needed by the *c_mode* field of the *cpio* archive format are described as follows:

Name	Description	Value (Octal)
C_IRUSR	Read by owner.	0000400
C_IWUSR	Write by owner.	0000200
C_IXUSR	Execute by owner.	0000100
C_IRGRP	Read by group.	0000040
C_IWGRP	Write by group.	0000020
C_IXGRP	Execute by group.	0000010
C_IROTH	Read by others.	0000004
C_IWOTH	Write by others.	0000002
C_IXOTH	Execute by others.	0000001
C_ISUID	Set user ID.	0004000
C_ISGID	Set group ID.	0002000
C_ISVTX	On directories, restricted deletion flag.	0001000
C_ISDIR	Directory.	0040000
C_ISFIFO	FIFO.	0010000
C_ISREG	Regular file.	0100000
C_ISBLK	Block special.	0060000
C_ISCHR	Character special.	0020000
C_ISCTG	Reserved.	0110000
C_ISLNK	Symbolic link.	0120000
C_ISSOCK	Socket.	0140000

The header shall define the symbolic constant:

MAGIC "070707"

The following sections are informative.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Shell and Utilities volume of IEEE Std 1003.1-2001, *pax*

COPYRIGHT

_tar.h>

NAME

tar.h – extended tar definitions

SYNOPSIS

#include <tar.h>

DESCRIPTION

The <*tar.h*> header shall define header block definitions as follows.

General definitions:

Name	Description	Value
TMAGIC	"ustar"	ustar plus null byte.
TMAGLEN	6	Length of the above.
TVERSION	"00"	00 without a null byte.
TVERSLEN	2	Length of the above.

Typeflag field definitions:

Name	Description	Value
REGTYPE	'0'	Regular file.
AREGTYPE	'\0'	Regular file.
LNKTYPE	'1'	Link.
SYMTYPE	'2'	Symbolic link.
CHRTYPE	'3'	Character special.
BLKTYPE	'4'	Block special.
DIRTYPE	'5'	Directory.
FIFOTYPE	'6'	FIFO special.
CONTTYPE	'7'	Reserved.

Mode field bit definitions (octal):

Name	Description	Value
TSUID	04000	Set UID on execution.
TSGID	02000	Set GID on execution.
TSVTX	01000	On directories, restricted deletion flag.
TUREAD	00400	Read by owner.
TUWRITE	00200	Write by owner special.
TUEXEC	00100	Execute/search by owner.
TGREAD	00040	Read by group.
TGWRITE	00020	Write by group.
TGEXEC	00010	Execute/search by group.
TOREAD	00004	Read by other.
TOWRITE	00002	Write by other.
TOEXEC	00001	Execute/search by other.

The following sections are informative.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The Shell and Utilities volume of IEEE Std 1003.1-2001, *pax*

COPYRIGHT

http://www.opengroup.org/unix/online.html . BSDCPIO

NAME

cpio — copy files to and from archives

SYNOPSIS

cpio { **-i** } [*options*] [*pattern ...*] [< *archive*]
cpio { **-o** } [*options*] < *name-list* [> *archive*]
cpio { **-p** } [*options*] *dest-dir* < *name-list*

DESCRIPTION

cpio copies files between archives and directories. This implementation can extract from tar, pax, cpio, zip, jar, ar, and ISO 9660 cdrom images and can create tar, pax, cpio, ar, and shar archives.

The first option to **cpio** is a mode indicator from the following list:

-i Input. Read an archive from standard input and extract the contents to disk or (if the **-t** option is specified) list the contents to standard output. If one or more file patterns are specified, only files matching one of the patterns will be extracted.

-o Output. Read a list of filenames from standard input and produce a new archive on standard output containing the specified items.

-p Pass-through. Read a list of filenames from standard input and copy the files to the specified directory.

OPTIONS

Unless specifically stated otherwise, options are applicable in all operating modes.

-a (o and p modes) Reset access times on files after they are read.

-B (o mode only) Block output to records of 5120 bytes.

-c (o mode only) Use the old POSIX portable character format. Equivalent to **--format** *odc*.

-d (i and p modes) Create directories as necessary.

-f *pattern*
(i mode only) Ignore files that match *pattern*.

--format *format*
(o mode only) Produce the output archive in the specified format. Supported formats include:

cpio	Synonym for *odc*.
newc	The SVR4 portable cpio format.
odc	The old POSIX.1 portable octet-oriented cpio format.
pax	The POSIX.1 pax format, an extension of the ustar format.
ustar	The POSIX.1 tar format.

The default format is *odc*. See `libarchive_formats`(5) for more complete information about the formats currently supported by the underlying `libarchive`(3) library.

-i Input mode. See above for description.

-L (o and p modes) All symbolic links will be followed. Normally, symbolic links are archived and copied as symbolic links. With this option, the target of the link will be archived or copied instead.

-l (p mode only) Create links from the target directory to the original files, instead of copying.

-m (i and p modes) Set file modification time on created files to match those in the source.

-o Output mode. See above for description.

-p Pass-through mode. See above for description.

--quiet
Suppress unnecessary messages.

-R [user][:][group]
Set the owner and/or group on files in the output. If group is specified with no user (for example, **-R** *:wheel*) then the group will be set but not the user. If the user is specified with a trailing colon and no group (for example, **-R** *root:*) then the group will be set to the user's default group. If the user is specified with no trailing colon, then the user will be set but not the group. In **-i** and **-p** modes, this option can only be used by the super-user. (For compatibility, a period can be used in place of the colon.)

-r (All modes.) Rename files interactively. For each file, a prompt is written to `/dev/tty` containing the name of the file and a line is read from `/dev/tty`. If the line read is blank, the file is skipped. If the line contains a single period, the file is processed normally. Otherwise, the line is taken to be the new name of the file.

-t (i mode only) List the contents of the archive to stdout; do not restore the contents to disk.

-u (i and p modes) Unconditionally overwrite existing files. Ordinarily, an older file will not overwrite a newer file on disk.

-v Print the name of each file to stderr as it is processed. With **-t**, provide a detailed listing of each file.

--version
Print the program version information and exit.

-y (o mode only) Compress the archive with bzip2-compatible compression before writing to stdout. In input mode, this option is ignored; bzip2 compression is recognized automatically on input.

-z (o mode only) Compress the archive with gzip-compatible compression before writing it to stdout. In input mode, this option is ignored; gzip compression is recognized automatically on input.

ENVIRONMENT

The following environment variables affect the execution of **cpio**:

`LANG` The locale to use. See `environ`(7) for more information.

`TZ` The timezone to use when displaying dates. See `environ`(7) for more information.

EXIT STATUS

The **cpio** utility exits 0 on success, and >0 if an error occurs.

EXAMPLES

The **cpio** command is traditionally used to copy file heirarchies in conjunction with the `find`(1) command. The first example here simply copies all files from `src` to `dest`:

```
find src | cpio -pmud dest
```

By carefully selecting options to the `find`(1) command and combining it with other standard utilities, it is possible to exercise very fine control over which files are copied. This next example copies files from `src` to `dest` that are more than 2 days old and whose names match a particular pattern:

```
find src -mtime +2 | grep foo[bar] | cpio -pdmu dest
```

This example copies files from `src` to `dest` that are more than 2 days old and which contain the word "foobar":

```
find src -mtime +2 | xargs grep -l foobar | cpio -pdmu dest
```

COMPATIBILITY

The mode options i, o, and p and the options a, B, c, d, f, l, m, r, t, u, and v comply with SUSv2.

The old POSIX.1 standard specified that only **-i**, **-o**, and **-p** were interpreted as command-line options. Each took a single argument of a list of modifier characters. For example, the standard syntax allows **-imu** but does not support **-miu** or **-i -m -u**, since *m* and *u* are only modifiers to **-i**, they

are not command-line options in their own right. The syntax supported by this implementation is backwards-compatible with the standard. For best compatibility, scripts should limit themselves to the standard syntax.

SEE ALSO

bzip2(1), tar(1), gzip(1), mt(1), pax(1), libarchive(3), cpio(5), libarchive-formats(5), tar(5)

STANDARDS

There is no current POSIX standard for the cpio command; it appeared in ISO/IEC 9945-1:1996 ("POSIX.1") but was dropped from IEEE Std 1003.1-2001 ("POSIX.1").

The cpio, ustar, and pax interchange file formats are defined by IEEE Std 1003.1-2001 ("POSIX.1") for the pax command.

HISTORY

The original **cpio** and **find** utilities were written by Dick Haight while working in AT&T's Unix Support Group. They first appeared in 1977 in PWB/UNIX 1.0, the "Programmer's Work Bench" system developed for use within AT&T. They were first released outside of AT&T as part of System III Unix in 1981. As a result, **cpio** actually predates **tar**, even though it was not well-known outside of AT&T until some time later.

This is a complete re-implementation based on the libarchive(3) library.

BUGS

The cpio archive format has several basic limitations: It does not store user and group names, only numbers. As a result, it cannot be reliably used to transfer files between systems with dissimilar user and group numbering. Older cpio formats limit the user and group numbers to 16 or 18 bits, which is insufficient for modern systems. The cpio archive formats cannot support files over 4 gigabytes, except for the "odc" variant, which can support files up to 8 gigabytes. BSDTAR

NAME

tar — manipulate tape archives

SYNOPSIS

tar [*bundled-flags* ⟨args⟩] [⟨*file*⟩ | ⟨*pattern*⟩ ...]
tar {**-c**} [*options*] [*files* | *directories*]
tar {**-r** | **-u**} **-f** *archive-file* [*options*] [*files* | *directories*]
tar {**-t** | **-x**} [*options*] [*patterns*]

DESCRIPTION

tar creates and manipulates streaming archive files. This implementation can extract from tar, pax, cpio, zip, jar, ar, and ISO 9660 cdrom images and can create tar, pax, cpio, ar, and shar archives.

The first synopsis form shows a "bundled" option word. This usage is provided for compatibility with historical implementations. See COMPATIBILITY below for details.

The other synopsis forms show the preferred usage. The first option to **tar** is a mode indicator from the following list:

-c Create a new archive containing the specified items.

-r Like **-c**, but new entries are appended to the archive. Note that this only works on uncompressed archives stored in regular files. The **-f** option is required.

-t List archive contents to stdout.

-u Like **-r**, but new entries are added only if they have a modification date newer than the corresponding entry in the archive. Note that this only works on uncompressed archives stored in regular files. The **-f** option is required.

-x Extract to disk from the archive. If a file with the same name appears more than once in the archive, each copy will be extracted, with later copies overwriting (replacing) earlier copies.

In **-c**, **-r**, or **-u** mode, each specified file or directory is added to the archive in the order specified on the command line. By default, the contents of each directory are also archived.

In extract or list mode, the entire command line is read and parsed before the archive is opened. The pathnames or patterns on the command line indicate which items in the archive should be processed. Patterns are shell-style globbing patterns as documented in `tcsh`(1).

OPTIONS

Unless specifically stated otherwise, options are applicable in all operating modes.

@archive
(c and r mode only) The specified archive is opened and the entries in it will be appended to the current archive. As a simple example,

```
tar -c -f - newfile @original.tar
```

writes a new archive to standard output containing a file `newfile` and all of the entries from `original.tar`. In contrast,

```
tar -c -f - newfile original.tar
```

creates a new archive with only two entries. Similarly,

```
tar -czf - --format pax @-
```

reads an archive from standard input (whose format will be determined automatically) and converts it into a gzip-compressed pax-format archive on stdout. In this way, **tar** can be used to convert archives from one format to another.

-b *blocksize*
Specify the block size, in 512-byte records, for tape drive I/O. As a rule, this argument is only needed when reading from or writing to tape drives, and usually not even then as the default block size of 20 records (10240 bytes) is very common.

-C *directory*
In c and r mode, this changes the directory before adding the following files. In x mode, change directories after opening the archive but before extracting entries from the archive.

--check-links (**-W check-links**)
(c and r modes only) Issue a warning message unless all links to each file are archived.

--exclude *pattern* (**-W exclude**=*pattern*)
Do not process files or directories that match the specified pattern. Note that exclusions take precedence over patterns or filenames specified on the command line.

--format *format* (**-W format**=*format*)
(c mode only) Use the specified format for the created archive. Supported formats include "cpio", "pax", "shar", and "ustar". Other formats may also be supported; see libarchive-formats(5) for more information about currently-supported formats.

-f *file*
Read the archive from or write the archive to the specified file. The filename can be – for standard input or standard output. If not specified, the default tape device will be used. (On FreeBSD, the default tape device is /dev/sa0.)

--fast-read (**-W fast-read**)
(x and t mode only) Extract or list only the first archive entry that matches each pattern or filename operand. Exit as soon as each specified pattern or filename has been matched. By default, the archive is always read to the very end, since there can be multiple entries with the same name and, by convention, later entries overwrite earlier entries. This option is provided as a performance optimization.

-H (c and r mode only) Symbolic links named on the command line will be followed; the target of the link will be archived, not the link itself.

-h (c and r mode only) Synonym for **-L**.

-I Synonym for **-T**.

--include *pattern* (**-W include**=*pattern*)
Process only files or directories that match the specified pattern. Note that exclusions specified with **--exclude** take precedence over inclusions. If no inclusions are explicitly specified, all entries are processed by default. The **--include** option is especially useful when filtering archives. For example, the command

tar -c -f new.tar **--include='*foo*' @**old.tgz

creates a new archive new.tar containing only the entries from old.tgz containing the string 'foo'.

-j (c mode only) Compress the resulting archive with bzip2(1). In extract or list modes, this option is ignored. Note that, unlike other **tar** implementations, this implementation recognizes bzip2 compression automatically when reading archives.

-k (x mode only) Do not overwrite existing files. In particular, if a file appears more than once in an archive, later copies will not overwrite earlier copies.

-L (c and r mode only) All symbolic links will be followed. Normally, symbolic links are archived as such. With this option, the target of the link will be archived instead.

-l This is a synonym for the **--check-links** option.

-m (x mode only) Do not extract modification time. By default, the modification time is set to the time stored in the archive.

-n (c, r, u modes only) Do not recursively archive the contents of directories.

--newer *date* (**-W newer**=*date*)
(c, r, u modes only) Only include files and directories newer than the specified date. This compares ctime entries.

--newer-mtime *date* (**-W newer-mtime**=*date*)
(c, r, u modes only) Like **--newer**, except it compares mtime entries instead of ctime entries.

--newer-than file (**-W newer-than**=file)
(c, r, u modes only) Only include files and directories newer than the specified file. This compares ctime entries.

--newer-mtime-than file (**-W newer-mtime-than**=file)
(c, r, u modes only) Like **--newer-than**, except it compares mtime entries instead of ctime entries.

--nodump (**-W nodump**)
(c and r modes only) Honor the nodump file flag by skipping this file.

--null (**-W null**)
(use with **-I**, **-T**, or **-X**) Filenames or patterns are separated by null characters, not by newlines. This is often used to read filenames output by the **-print0** option to find(1).

-O (x, t modes only) In extract (-x) mode, files will be written to standard out rather than being extracted to disk. In list (-t) mode, the file listing will be written to stderr rather than the usual stdout.

-o (x mode only) Use the user and group of the user running the program rather than those specified in the archive. Note that this has no significance unless **-p** is specified, and the program is being run by the root user. In this case, the file modes and flags from the archive will be restored, but ACLs or owner information in the archive will be discarded.

--one-file-system (**-W one-file-system**)
(c, r, and u modes) Do not cross mount points.

-P Preserve pathnames. By default, absolute pathnames (those that begin with a / character) have the leading slash removed both when creating archives and extracting from them. Also, **tar** will refuse to extract archive entries whose pathnames contain **..** or whose target directory would be altered by a symlink. This option suppresses these behaviors.

-p (x mode only) Preserve file permissions. Attempt to restore the full permissions, including owner, file modes, file flags and ACLs, if available, for each item extracted from the archive. By default, newly-created files are owned by the user running **tar**, the file mode is restored for newly-created regular files, and all other types of entries receive default permissions. If **tar** is being run by root, the default is to restore the owner unless the **-o** option is also specified.

--strip-components *count* (**-W strip-components**=*count*)
(x and t mode only) Remove the specified number of leading path elements. Pathnames with fewer elements will be silently skipped. Note that the pathname is edited after checking inclusion/exclusion patterns but before security checks.

-T *filename*
In x or t mode, **tar** will read the list of names to be extracted from filename. In c mode, **tar** will read names to be archived from filename. The special name "-C" on a line by itself will cause the current directory to be changed to the directory specified on the following line. Names are terminated by newlines unless **--null** is specified. Note that **--null** also disables the special handling of lines containing "-C".

-U (x mode only) Unlink files before creating them. Without this option, **tar** overwrites existing files, which preserves existing hardlinks. With this option, existing hardlinks will be broken, as will any symlink that would affect the location of an extracted file.

--use-compress-program *program*
Pipe the input (in x or t mode) or the output (in c mode) through program instead of using the builtin compression support.

-v Produce verbose output. In create and extract modes, **tar** will list each file name as it is read from or written to the archive. In list mode, **tar** will produce output similar to that of ls(1). Additional **-v** options will provide additional detail.

-W *longopt=value*
Long options (preceded by **--**) are only supported directly on systems that have the getopt_long(3) function. The **-W** option can be used to access long options on systems that do not support this function.

-w Ask for confirmation for every action.

-X *filename*
Read a list of exclusion patterns from the specified file. See **--exclude** for more information about the handling of exclusions.

-y (c mode only) Compress the resulting archive with bzip2(1). In extract or list modes, this option is ignored. Note that, unlike other **tar** implementations, this implementation recognizes bzip2 compression automatically when reading archives.

-z (c mode only) Compress the resulting archive with gzip(1). In extract or list modes, this option is ignored. Note that, unlike other **tar** implementations, this implementation recognizes gzip compression automatically when reading archives.

ENVIRONMENT

The following environment variables affect the execution of **tar**:

LANG The locale to use. See environ(7) for more information.

TAPE The default tape device. The **-f** option overrides this.

TZ The timezone to use when displaying dates. See environ(7) for more information.

FILES

/dev/sa0 The default tape device, if not overridden by the TAPE environment variable or the **-f** option.

EXIT STATUS

The **tar** utility exits 0 on success, and >0 if an error occurs.

EXAMPLES

The following creates a new archive called *file.tar.gz* that contains two files *source.c* and *source.h*:

```
tar -czf file.tar.gz source.c source.h
```

To view a detailed table of contents for this archive:

```
tar -tvf file.tar.gz
```

To extract all entries from the archive on the default tape drive:

```
tar -x
```

To examine the contents of an ISO 9660 cdrom image:

```
tar -tf image.iso
```

To move file hierarchies, invoke **tar** as

```
tar -cf - -C srcdir . | tar -xpf - -C destdir
```

or more traditionally

```
cd srcdir ; tar -cf - . | (cd destdir ; tar -xpf -)
```

In create mode, the list of files and directories to be archived can also include directory change instructions of the form **-C**foo/baz and archive inclusions of the form **@**archive-file. For example, the command line

```
tar -c -f new.tar foo1 @old.tgz -C/tmp foo2
```

will create a new archive new.tar. **tar** will read the file foo1 from the current directory and add it to the output archive. It will then read each entry from old.tgz and add those entries to the output archive. Finally, it will switch to the /tmp directory and add foo2 to the output archive.

The **--newer** and **--newer-mtime** switches accept a variety of common date and time specifications, including “12 Mar 2005 7:14:29pm”, “2005-03-12 19:14”, “5 minutes ago”, and “19:14 PST May 1”.

COMPATIBILITY

The bundled-arguments format is supported for compatibility with historic implementations. It consists of an initial word (with no leading - character) in which each character indicates an option. Arguments follow as separate words. The order of the arguments must match the order of the corresponding characters in the bundled command word. For example,

tar tbf 32 file.tar

specifies three flags **t**, **b**, and **f**. The **b** and **f** flags both require arguments, so there must be two additional items on the command line. The *32* is the argument to the **b** flag, and *file.tar* is the argument to the **f** flag.

The mode options c, r, t, u, and x and the options b, f, l, m, o, v, and w comply with SUSv2.

For maximum portability, scripts that invoke **tar** should use the bundled-argument format above, should limit themselves to the **c**, **t**, and **x** modes, and the **b**, **f**, **m**, **v**, and **w** options.

On systems that support getopt_long(), additional long options are available to improve compatibility with other tar implementations.

SECURITY

Certain security issues are common to many archiving programs, including **tar**. In particular, carefully-crafted archives can request that **tar** extract files to locations outside of the target directory. This can potentially be used to cause unwitting users to overwrite files they did not intend to overwrite. If the archive is being extracted by the superuser, any file on the system can potentially be overwritten. There are three ways this can happen. Although **tar** has mechanisms to protect against each one, savvy users should be aware of the implications:

- Archive entries can have absolute pathnames. By default, **tar** removes the leading / character from filenames before restoring them to guard against this problem.
- Archive entries can have pathnames that include **..** components. By default, **tar** will not extract files containing **..** components in their pathname.
- Archive entries can exploit symbolic links to restore files to other directories. An archive can restore a symbolic link to another directory, then use that link to restore a file into that directory. To guard against this, **tar** checks each extracted path for symlinks. If the final path element is a symlink, it will be removed and replaced with the archive entry. If **-U** is specified, any intermediate symlink will also be unconditionally removed. If neither **-U** nor **-P** is specified, **tar** will refuse to extract the entry.

To protect yourself, you should be wary of any archives that come from untrusted sources. You should examine the contents of an archive with

tar -tf filename

before extraction. You should use the **-k** option to ensure that **tar** will not overwrite any existing files or the **-U** option to remove any pre-existing files. You should generally not extract archives while running with super-user privileges. Note that the **-P** option to **tar** disables the security checks above and allows you to extract an archive while preserving any absolute pathnames, **..** components, or symlinks to other directories.

SEE ALSO

bzip2(1), cpio(1), gzip(1), mt(1), pax(1), shar(1), libarchive(3), libarchive-formats(5), tar(5)

STANDARDS

There is no current POSIX standard for the tar command; it appeared in ISO/IEC 9945-1:1996 (“POSIX.1”) but was dropped from IEEE Std 1003.1-2001 (“POSIX.1”). The options used by this implementation were developed by surveying a number of existing tar implementations as well as the old

POSIX specification for tar and the current POSIX specification for pax.

The ustar and pax interchange file formats are defined by IEEE Std 1003.1-2001 (“POSIX.1”) for the pax command.

HISTORY

A **tar** command appeared in Seventh Edition Unix, which was released in January, 1979. There have been numerous other implementations, many of which extended the file format. John Gilmore's **pdtar** public-domain implementation (circa November, 1987) was quite influential, and formed the basis of GNU tar. GNU tar was included as the standard system tar in FreeBSD beginning with FreeBSD 1.0.

This is a complete re-implementation based on the libarchive(3) library.

BUGS

This program follows ISO/IEC 9945-1:1996 (“POSIX.1”) for the definition of the **-l** option. Note that GNU tar prior to version 1.15 treated **-l** as a synonym for the **--one-file-system** option.

The **-C** dir option may differ from historic implementations.

All archive output is written in correctly-sized blocks, even if the output is being compressed. Whether or not the last output block is padded to a full block size varies depending on the format and the output device. For tar and cpio formats, the last block of output is padded to a full block size if the output is being written to standard output or to a character or block device such as a tape drive. If the output is being written to a regular file, the last block will not be padded. Many compressors, including gzip(1) and bzip2(1), complain about the null padding when decompressing an archive created by **tar**, although they still extract it correctly.

The compression and decompression is implemented internally, so there may be insignificant differences between the compressed output generated by

tar -czf - file

and that generated by

tar -cf - file | **gzip**

The default should be to read and write archives to the standard I/O paths, but tradition (and POSIX) dictates otherwise.

The **r** and **u** modes require that the archive be uncompressed and located in a regular file on disk. Other archives can be modified using **c** mode with the @archive-file extension.

To archive a file called @foo or -foo you must specify it as ./@foo or ./-foo, respectively.

In create mode, a leading ./ is always removed. A leading / is stripped unless the **-P** option is specified.

There needs to be better support for file selection on both create and extract.

There is not yet any support for multi-volume archives or for archiving sparse files.

Converting between dissimilar archive formats (such as tar and cpio) using the **@**- convention can cause hard link information to be lost. (This is a consequence of the incompatible ways that different archive formats store hardlink information.)

There are alternative long options for many of the short options that are deliberately not documented.

AR

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

ar – create and maintain library archives

SYNOPSIS

ar -d[-v] *archive file* **...**

ar -m [-v] *archive file* **...**

ar -m -a[-v] *posname archive file* **...**

ar -m -b[-v] *posname archive file* **...**

ar -m -i[-v] *posname archive file* **...**

ar -p[-v][-s]*archive* **[***file* **...]**

ar -q[-cv] *archive file* **...**

ar -r[-cuv] *archive file* **...**

ar -r -a[-cuv] *posname archive file* **...**

ar -r -b[-cuv] *posname archive file* **...**

ar -r -i[-cuv] *posname archive file* **...**

ar -t[-v][-s]*archive* **[***file* **...]**

ar -x[-v][-sCT]*archive* **[***file* **...]**

DESCRIPTION

The *ar* utility is part of the Software Development Utilities option.

The *ar* utility can be used to create and maintain groups of files combined into an archive. Once an archive has been created, new files can be added, and existing files in an archive can be extracted, deleted, or replaced. When an archive consists entirely of valid object files, the implementation shall format the archive so that it is usable as a library for link editing (see *c99* and *fort77*). When some of the archived files are not valid object files, the suitability of the archive for library use is undefined. If an archive consists entirely of printable files, the entire archive shall be printable.

When *ar* creates an archive, it creates administrative information indicating whether a symbol table is present in the archive. When there is at least one object file that *ar* recognizes as such in the archive, an archive symbol table shall be created in the archive and maintained by *ar*; it is used by the link editor to search the archive. Whenever the *ar* utility is used to create or update the contents of such an archive, the symbol table shall be rebuilt. The **-s** option shall force the symbol table to be rebuilt.

All *file* operands can be pathnames. However, files within archives shall be named by a filename, which is the last component of the pathname used when the file was entered into the archive. The comparison of *file* operands to the names of files in archives shall be performed by comparing the last component of the operand to the name of the file in the archive.

It is unspecified whether multiple files in the archive may be identically named. In the case of such files, however, each *file* and *posname* operand shall match only the first file in the archive having a name that is the same as the last component of the operand.

OPTIONS

The *ar* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax Guidelines.

The following options shall be supported:

-a Position new files in the archive after the file named by the *posname* operand.

-b Position new files in the archive before the file named by the *posname* operand.

-c Suppress the diagnostic message that is written to standard error by default when the archive *archive* is created.

-C Prevent extracted files from replacing like-named files in the file system. This option is useful when **-T** is also used, to prevent truncated filenames from replacing files with the same prefix.

-d Delete one or more *file*s from *archive*.

-i Position new files in the archive before the file in the archive named by the *posname* operand (equivalent to **-b**).

-m Move the named files in the archive. The **-a**, **-b**, or **-i** options with the *posname* operand indicate the position; otherwise, move the names files in the archive to the end of the archive.

-p Write the contents of the *file*s in the archive named by *file* operands from *archive* to the standard output. If no *file* operands are specified, the contents of all files in the archive shall be written in the order of the archive.

-q Append the named files to the end of the archive. In this case *ar* does not check whether the added files are already in the archive. This is useful to bypass the searching otherwise done when creating a large archive piece by piece.

-r Replace or add *file*s to *archive*. If the archive named by *archive* does not exist, a new archive shall be created and a diagnostic message shall be written to standard error (unless the **-c** option is specified). If no *file*s are specified and the *archive* exists, the results are undefined. Files that replace existing files in the archive shall not change the order of the archive. Files that do not replace existing files in the archive shall be appended to the archive unless a **-a**, **-b**, or **-i** option specifies another position.

-s Force the regeneration of the archive symbol table even if *ar* is not invoked with an option that modifies the archive contents. This option is useful to restore the archive symbol table after it has been stripped; see *strip*.

-t Write a table of contents of *archive* to the standard output. The files specified by the *file* operands shall be included in the written list. If no *file* operands are specified, all files in *archive* shall be included in the order of the archive.

-T Allow filename truncation of extracted files whose archive names are longer than the file system can support. By default, extracting a file with a name that is too long shall be an error; a diagnostic message shall be written and the file shall not be extracted.

-u Update older files in the archive. When used with the **-r** option, files in the archive shall be replaced only if the corresponding *file* has a modification time that is at least as new as the modification time of the file in the archive.

-v Give verbose output. When used with the option characters **-d**, **-r**, or **-x**, write a detailed file-by-file description of the archive creation and maintenance activity, as described in the STDOUT section.

When used with **-p**, write the name of the file in the archive to the standard output before writing the file in the archive itself to the standard output, as described in the STDOUT section.

When used with **-t**, include a long listing of information about the files in the archive, as described in the STDOUT section.

-x Extract the files in the archive named by the *file* operands from *archive*. The contents of the archive shall not be changed. If no *file* operands are given, all files in the archive shall be

extracted. The modification time of each file extracted shall be set to the time the file is extracted from the archive.

OPERANDS

The following operands shall be supported:

archive A pathname of the archive.

file A pathname. Only the last component shall be used when comparing against the names of files in the archive. If two or more *file* operands have the same last pathname component (basename), the results are unspecified. The implementation's archive format shall not truncate valid filenames of files added to or replaced in the archive.

posname
The name of a file in the archive, used for relative positioning; see options **-m** and **-r**.

STDIN

Not used.

INPUT FILES

The archive named by *archive* shall be a file in the format created by *ar* **-r**.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *ar*:

LANG Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL
If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the format and content for date and time strings written by *ar* **-tv**.

NLSPATH
Determine the location of message catalogs for the processing of *LC_MESSAGES* .

TMPDIR
Determine the pathname that overrides the default directory for temporary files, if any.

TZ Determine the timezone used to calculate date and time strings written by *ar* **-tv**. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the **-d** option is used with the **-v** option, the standard output format shall be:

"d - %s\n", <*file***>**

where *file* is the operand specified on the command line.

If the **-p** option is used with the **-v** option, *ar* shall precede the contents of each file with:

"\n<%s>\n\n", <*file***>**

where *file* is the operand specified on the command line, if *file* operands were specified, and the name

of the file in the archive if they were not.

If the **-r** option is used with the **-v** option:

* If *file* is already in the archive, the standard output format shall be:

  "r - %s\n", <*file***>**

where <*file*> is the operand specified on the command line.

* If *file* is not already in the archive, the standard output format shall be:

  "a - %s\n", <*file***>**

where <*file*> is the operand specified on the command line.

If the **-t** option is used, *ar* shall write the names of the files in the archive to the standard output in the format:

"%s\n", <*file***>**

where *file* is the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

If the **-t** option is used with the **-v** option, the standard output format shall be:

"%s %u/%u %u %s %d %d:%d %d %s\n", <*member mode***>, <***user ID***>,**
<*group ID***>, <***number of bytes in member***>,**
<*abbreviated month***>, <***day-of-month***>, <***hour***>,**
<*minute***>, <***year***>, <***file***>**

where:

<*file*> Shall be the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

<*member mode*>

Shall be formatted the same as the <*file mode*> string defined in the STDOUT section of *ls*, except that the first character, the <*entry type*>, is not used; the string represents the file mode of the file in the archive at the time it was added to or replaced in the archive.

The following represent the last-modification time of a file when it was most recently added to or replaced in the archive:

<*abbreviated month*>

Equivalent to the format of the **%b** conversion specification format in *date*.

<*day-of-month*>

Equivalent to the format of the **%e** conversion specification format in *date*.

<*hour*>
Equivalent to the format of the **%H** conversion specification format in *date*.

<*minute*>
Equivalent to the format of the **%M** conversion specification format in *date*.

<*year*> Equivalent to the format of the **%Y** conversion specification format in *date*.

When *LC_TIME* does not specify the POSIX locale, a different format and order of presentation of these fields relative to each other may be used in a format appropriate in the specified locale.

If the **-x** option is used with the **-v** option, the standard output format shall be:

"x - %s\n", <*file***>**

where *file* is the operand specified on the command line, if *file* operands were specified, or the name of

the file in the archive if they were not.

STDERR

The standard error shall be used only for diagnostic messages. The diagnostic message about creating a new archive when **-c** is not specified shall not modify the exit status.

OUTPUT FILES

Archives are files with unspecified formats.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values shall be returned:

0 Successful completion.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

The following sections are informative.

APPLICATION USAGE

None.

EXAMPLES

None.

RATIONALE

The archive format is not described. It is recognized that there are several known *ar* formats, which are not compatible. The *ar* utility is included, however, to allow creation of archives that are intended for use only on one machine. The archive is specified as a file, and it can be moved as a file. This does allow an archive to be moved from one machine to another machine that uses the same implementation of *ar*.

Utilities such as *pax* (and its forebears *tar* and *cpio*) also provide portable "archives". This is a not a duplication; the *ar* utility is included to provide an interface primarily for *make* and the compilers, based on a historical model.

In historical implementations, the **-q** option (available on XSI-conforming systems) is known to execute quickly because *ar* does not check on whether the added members are already in the archive. This is useful to bypass the searching otherwise done when creating a large archive piece-by-piece. These remarks may but need not remain true for a brand new implementation of this utility; hence, these remarks have been moved into the RATIONALE.

BSD implementations historically required applications to provide the **-s** option whenever the archive was supposed to contain a symbol table. As in this volume of IEEE Std 1003.1-2001, System V historically creates or updates an archive symbol table whenever an object file is removed from, added to, or updated in the archive.

The OPERANDS section requires what might seem to be true without specifying it: the archive cannot truncate the filenames below {NAME_MAX}. Some historical implementations do so, however, causing unexpected results for the application. Therefore, this volume of IEEE Std 1003.1-2001 makes the requirement explicit to avoid misunderstandings.

According to the System V documentation, the options **-dmpqrtx** are not required to begin with a hyphen (**'-'**). This volume of IEEE Std 1003.1-2001 requires that a conforming application use the leading hyphen.

The archive format used by the 4.4 BSD implementation is documented in this RATIONALE as an example: A file created by *ar* begins with the "magic" string **"!<arch>\n"** . The rest of the archive is made up of objects, each of which is composed of a header for a file, a possible filename, and the file contents. The header is portable between machine architectures, and, if the file contents are printable, the archive is itself printable.

The header is made up of six ASCII fields, followed by a two-character trailer. The fields are the object

name (16 characters), the file last modification time (12 characters), the user and group IDs (each 6 characters), the file mode (8 characters), and the file size (10 characters). All numeric fields are in decimal, except for the file mode, which is in octal.

The modification time is the file *st_mtime* field. The user and group IDs are the file *st_uid* and *st_gid* fields. The file mode is the file *st_mode* field. The file size is the file *st_size* field. The two-byte trailer is the string **"`<newline>"** .

Only the name field has any provision for overflow. If any filename is more than 16 characters in length or contains an embedded space, the string **"#1/"** followed by the ASCII length of the name is written in the name field. The file size (stored in the archive header) is incremented by the length of the name. The name is then written immediately following the archive header.

Any unused characters in any of these fields are written as <space>s. If any fields are their particular maximum number of characters in length, there is no separation between the fields.

Objects in the archive are always an even number of bytes long; files that are an odd number of bytes long are padded with a <newline>, although the size in the header does not reflect this.

The *ar* utility description requires that (when all its members are valid object files) *ar* produce an object code library, which the linkage editor can use to extract object modules. If the linkage editor needs a symbol table to permit random access to the archive, *ar* must provide it; however, *ar* does not require a symbol table.

The BSD **-o** option was omitted. It is a rare conforming application that uses *ar* to extract object code from a library with concern for its modification time, since this can only be of importance to *make*. Hence, since this functionality is not deemed important for applications portability, the modification time of the extracted files is set to the current time.

There is at least one known implementation (for a small computer) that can accommodate only object files for that system, disallowing mixed object and other files. The ability to handle any type of file is not only historical practice for most implementations, but is also a reasonable expectation.

Consideration was given to changing the output format of *ar* **-tv** to the same format as the output of *ls* **-l**. This would have made parsing the output of *ar* the same as that of *ls*. This was rejected in part because the current *ar* format is commonly used and changes would break historical usage. Second, *ar* gives the user ID and group ID in numeric format separated by a slash. Changing this to be the user name and group name would not be correct if the archive were moved to a machine that contained a different user database. Since *ar* cannot know whether the archive was generated on the same machine, it cannot tell what to report.

The text on the **-ur** option combination is historical practice-since one filename can easily represent two different files (for example, **/a/foo** and **/b/foo**), it is reasonable to replace the file in the archive even when the modification time in the archive is identical to that in the file system.

FUTURE DIRECTIONS

None.

SEE ALSO

c99, *date*, *fort77*, *pax*, *strip* the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 13, Headers, <*unistd.h*> description of {POSIX_NO_TRUNC}

COPYRIGHT

PAX

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

pax − portable archive interchange

SYNOPSIS

pax [-cdnv][-H|-L][-f *archive*]**[-s** *replstr*]**...[**pattern**...]**

pax -r[-cdiknuv][-H|-L][-f *archive*]**[-o** *options*]**...[-p** *string*]**...**
[-s *replstr*]**...[**pattern**...]**

pax -w[-dituvX][-H|-L][-b *blocksize*]**[[-a][-f** *archive*]**[-o** *options*]**...**
[-s *replstr*]**...[-x** *format*]**[**file**...]**

pax -r -w[-diklntuvX][-H|-L][-p *string*]**...[-s** *replstr*]**...**
[file**...]** *directory*

DESCRIPTION

The *pax* utility shall read, write, and write lists of the members of archive files and copy directory hierarchies. A variety of archive formats shall be supported; see the **-x** *format* option.

The action to be taken depends on the presence of the **-r** and **-w** options. The four combinations of **-r** and **-w** are referred to as the four modes of operation: **list**, **read**, **write**, and **copy** modes, corresponding respectively to the four forms shown in the SYNOPSIS section.

list In **list** mode (when neither **-r** nor **-w** are specified), *pax* shall write the names of the members of the archive file read from the standard input, with pathnames matching the specified patterns, to standard output. If a named file is of type directory, the file hierarchy rooted at that file shall be listed as well.

read In **read** mode (when **-r** is specified, but **-w** is not), *pax* shall extract the members of the archive file read from the standard input, with pathnames matching the specified patterns. If an extracted file is of type directory, the file hierarchy rooted at that file shall be extracted as well. The extracted files shall be created performing pathname resolution with the directory in which *pax* was invoked as the current working directory.

If an attempt is made to extract a directory when the directory already exists, this shall not be considered an error. If an attempt is made to extract a FIFO when the FIFO already exists, this shall not be considered an error.

The ownership, access, and modification times, and file mode of the restored files are discussed under the **-p** option.

write In **write** mode (when **-w** is specified, but **-r** is not), *pax* shall write the contents of the *file* operands to the standard output in an archive format. If no *file* operands are specified, a list of files to copy, one per line, shall be read from the standard input. A file of type directory shall include all of the files in the file hierarchy rooted at the file.

copy In **copy** mode (when both **-r** and **-w** are specified), *pax* shall copy the *file* operands to the destination directory.

If no *file* operands are specified, a list of files to copy, one per line, shall be read from the standard input. A file of type directory shall include all of the files in the file hierarchy rooted at the file.

The effect of the **copy** shall be as if the copied files were written to an archive file and then subsequently extracted, except that there may be hard links between the original and the copied files. If the destination directory is a subdirectory of one of the files to be copied, the results are unspecified. If the destination directory is a file of a type not defined by the System Interfaces volume of IEEE Std 1003.1-2001, the results are implementation-defined; otherwise, it shall be an error for the file named by the *directory* operand not to exist, not be writable by the user, or not be a file of type

directory.

In **read** or **copy** modes, if intermediate directories are necessary to extract an archive member, *pax* shall perform actions equivalent to the *mkdir*() function defined in the System Interfaces volume of IEEE Std 1003.1-2001, called with the following arguments:

* The intermediate directory used as the *path* argument
* The value of the bitwise-inclusive OR of S_IRWXU, S_IRWXG, and S_IRWXO as the *mode* argument

If any specified *pattern* or *file* operands are not matched by at least one file or archive member, *pax* shall write a diagnostic message to standard error for each one that did not match and exit with a non-zero exit status.

The archive formats described in the EXTENDED DESCRIPTION section shall be automatically detected on input. The default output archive format shall be implementation-defined.

A single archive can span multiple files. The *pax* utility shall determine, in an implementation-defined manner, what file to read or write as the next file.

If the selected archive format supports the specification of linked files, it shall be an error if these files cannot be linked when the archive is extracted. For archive formats that do not store file contents with each name that causes a hard link, if the file that contains the data is not extracted during this *pax* session, either the data shall be restored from the original file, or a diagnostic message shall be displayed with the name of a file that can be used to extract the data. In traversing directories, *pax* shall detect infinite loops; that is, entering a previously visited directory that is an ancestor of the last file visited. When it detects an infinite loop, *pax* shall write a diagnostic message to standard error and shall terminate.

OPTIONS

The *pax* utility shall conform to the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, Utility Syntax Guidelines, except that the order of presentation of the **-o**, **-p**, and **-s** options is significant.

The following options shall be supported:

-r Read an archive file from standard input.

-w Write files to the standard output in the specified archive format.

-a Append files to the end of the archive. It is implementation-defined which devices on the system support appending. Additional file formats unspecified by this volume of IEEE Std 1003.1-2001 may impose restrictions on appending.

-b *blocksize*
Block the output at a positive decimal integer number of bytes per write to the archive file. Devices and archive formats may impose restrictions on blocking. Blocking shall be automatically determined on input. Conforming applications shall not specify a *blocksize* value larger than 32256. Default blocking when creating archives depends on the archive format. (See the **-x** option below.)

-c Match all file or archive members except those specified by the *pattern* or *file* operands.

-d Cause files of type directory being copied or archived or archive members of type directory being extracted or listed to match only the file or archive member itself and not the file hierarchy rooted at the file.

-f *archive*
Specify the pathname of the input or output archive, overriding the default standard input (in **list** or **read** modes) or standard output (**write** mode).

-H If a symbolic link referencing a file of type directory is specified on the command line, *pax* shall archive the file hierarchy rooted in the file referenced by the link, using the name of the link as the root of the file hierarchy. Otherwise, if a symbolic link referencing a file of any other file type which *pax* can normally archive is specified on the command line, then *pax* shall archive the file referenced by the link, using the name of the link. The default behavior

shall be to archive the symbolic link itself.

-i Interactively rename files or archive members. For each archive member matching a *pattern* operand or file matching a *file* operand, a prompt shall be written to the file **/dev/tty**. The prompt shall contain the name of the file or archive member, but the format is otherwise unspecified. A line shall then be read from **/dev/tty**. If this line is blank, the file or archive member shall be skipped. If this line consists of a single period, the file or archive member shall be processed with no modification to its name. Otherwise, its name shall be replaced with the contents of the line. The *pax* utility shall immediately exit with a non-zero exit status if end-of-file is encountered when reading a response or if **/dev/tty** cannot be opened for reading and writing.

The results of extracting a hard link to a file that has been renamed during extraction are unspecified.

-k Prevent the overwriting of existing files.

-l (The letter ell.) In **copy** mode, hard links shall be made between the source and destination file hierarchies whenever possible. If specified in conjunction with **-H** or **-L**, when a symbolic link is encountered, the hard link created in the destination file hierarchy shall be to the file referenced by the symbolic link. If specified when neither **-H** nor **-L** is specified, when a symbolic link is encountered, the implementation shall create a hard link to the symbolic link in the source file hierarchy or copy the symbolic link to the destination.

-L If a symbolic link referencing a file of type directory is specified on the command line or encountered during the traversal of a file hierarchy, *pax* shall archive the file hierarchy rooted in the file referenced by the link, using the name of the link as the root of the file hierarchy. Otherwise, if a symbolic link referencing a file of any other file type which *pax* can normally archive is specified on the command line or encountered during the traversal of a file hierarchy, *pax* shall archive the file referenced by the link, using the name of the link. The default behavior shall be to archive the symbolic link itself.

-n Select the first archive member that matches each *pattern* operand. No more than one archive member shall be matched for each pattern (although members of type directory shall still match the file hierarchy rooted at that file).

-o *options*

Provide information to the implementation to modify the algorithm for extracting or writing files. The value of *options* shall consist of one or more comma-separated keywords of the form:

keyword[**[:]=***value*][**,***keyword*[**[:]=***value*]**, ...]**

Some keywords apply only to certain file formats, as indicated with each description. Use of keywords that are inapplicable to the file format being processed produces undefined results.

Keywords in the *options* argument shall be a string that would be a valid portable filename as described in the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.276, Portable Filename Character Set.

Note:

Keywords are not expected to be filenames, merely to follow the same character composition rules as portable filenames.

Keywords can be preceded with white space. The *value* field shall consist of zero or more characters; within *value*, the application shall precede any literal comma with a backslash, which shall be ignored, but preserves the comma as part of *value*. A comma as the final character, or a comma followed solely by white space as the final characters, in *options* shall be ignored. Multiple **-o** options can be specified; if keywords given to these multiple **-o** options conflict, the keywords and values appearing later in command line sequence shall take precedence and the earlier shall be silently ignored. The following keyword values of *options* shall be supported for the file formats as indicated:

delete=*pattern*

(Applicable only to the **-x pax** format.) When used in **write** or **copy** mode, *pax* shall omit from extended header records that it produces any keywords matching the string pattern. When used in **read** or **list** mode, *pax* shall ignore any keywords matching the string pattern in the extended header records. In both cases, matching shall be performed using the pattern matching notation described in *Patterns Matching a Single Character* and *Patterns Matching Multiple Characters* . For example:

-o delete=*security***.***

would suppress security-related information. See pax Extended Header for extended header record keyword usage.

exthdr.name=*string*

(Applicable only to the **-x pax** format.) This keyword allows user control over the name that is written into the **ustar** header blocks for the extended header produced under the circumstances described in pax Header Block . The name shall be the contents of *string*, after the following character substitutions have been made:

string	
Includes:	**Replaced By:**
%d	The directory name of the file, equivalent to the result of the *dirname* utility on the translated pathname.
%f	The filename of the file, equivalent to the result of the *basename* utility on the translated pathname.
%p	The process ID of the *pax* process.
%%	A **'%'** character.

Any other **'%'** characters in *string* produce undefined results.

If no **-o exthdr.name=** *string* is specified, *pax* shall use the following default value:

%d/PaxHeaders.%p/%f

globexthdr.name=*string*

(Applicable only to the **-x pax** format.) When used in **write** or **copy** mode with the appropriate options, *pax* shall create global extended header records with **ustar** header blocks that will be treated as regular files by previous versions of *pax*. This keyword allows user control over the name that is written into the **ustar** header blocks for global extended header records. The name shall be the contents of string, after the following character substitutions have been made:

string	
Includes:	**Replaced By:**
%n	An integer that represents the sequence number of the global extended header record in the archive, starting at 1.
%p	The process ID of the *pax* process.
%%	A **'%'** character.

Any other **'%'** characters in *string* produce undefined results.

If no **-o globexthdr.name=** *string* is specified, *pax* shall use the following default value:

$TMPDIR/GlobalHead.%p.%n

where $ *TMPDIR* represents the value of the *TMPDIR* environment variable. If *TMPDIR* is not set, *pax* shall use **/tmp**.

invalid=*action*

(Applicable only to the **-x pax** format.) This keyword allows user control over the action *pax* takes upon encountering values in an extended header record that, in **read** or **copy** mode, are

invalid in the destination hierarchy or, in **list** mode, cannot be written in the codeset and current locale of the implementation. The following are invalid values that shall be recognized by *pax*:

* In **read** or **copy** mode, a filename or link name that contains character encodings invalid in the destination hierarchy. (For example, the name may contain embedded NULs.)
* In **read** or **copy** mode, a filename or link name that is longer than the maximum allowed in the destination hierarchy (for either a pathname component or the entire pathname).
* In **list** mode, any character string value (filename, link name, user name, and so on) that cannot be written in the codeset and current locale of the implementation.

The following mutually-exclusive values of the *action* argument are supported:

bypass
In **read** or **copy** mode, *pax* shall bypass the file, causing no change to the destination hierarchy. In **list** mode, *pax* shall write all requested valid values for the file, but its method for writing invalid values is unspecified.

rename
In **read** or **copy** mode, *pax* shall act as if the **-i** option were in effect for each file with invalid filename or link name values, allowing the user to provide a replacement name interactively. In **list** mode, *pax* shall behave identically to the **bypass** action.

UTF-8
When used in **read**, **copy**, or **list** mode and a filename, link name, owner name, or any other field in an extended header record cannot be translated from the **pax** UTF-8 codeset format to the codeset and current locale of the implementation, *pax* shall use the actual UTF-8 encoding for the name.

write
In **read** or **copy** mode, *pax* shall write the file, translating or truncating the name, regardless of whether this may overwrite an existing file with a valid name. In **list** mode, *pax* shall behave identically to the **bypass** action.

If no **-o invalid=** option is specified, *pax* shall act as if **-o invalid= bypass** were specified. Any overwriting of existing files that may be allowed by the **-o invalid=** actions shall be subject to permission (**-p**) and modification time (**-u**) restrictions, and shall be suppressed if the **-k** option is also specified.

linkdata

(Applicable only to the **-x pax** format.) In **write** mode, *pax* shall write the contents of a file to the archive even when that file is merely a hard link to a file whose contents have already been written to the archive.

listopt=*format*

This keyword specifies the output format of the table of contents produced when the **-v** option is specified in **list** mode. See List Mode Format Specifications . To avoid ambiguity, the **listopt=** *format* shall be the only or final **keyword=** *value* pair in a **-o** option-argument; all characters in the remainder of the option-argument shall be considered part of the format string. When multiple **-o listopt=** *format* options are specified, the format strings shall be considered a single, concatenated string, evaluated in command line order.

times

(Applicable only to the **-x** *pax* format.) When used in **write** or **copy** mode, *pax* shall include **atime**, **ctime**, and **mtime** extended header records for each file. See pax Extended Header File Times .

In addition to these keywords, if the **-x** *pax* format is specified, any of the keywords and values defined in pax Extended Header, including implementation extensions, can be used in **-o** option-arguments, in either of two modes:

keyword=*value*

When used in **write** or **copy** mode, these keyword/value pairs shall be included at the beginning of the archive as **typeflag g** global extended header records. When used in **read** or **list** mode, these keyword/value pairs shall act as if they had been at the beginning of the archive as **typeflag g** global extended header records.

keyword:=*value*

When used in **write** or **copy** mode, these keyword/value pairs shall be included as records at the beginning of a **typeflag x** extended header for each file. (This shall be equivalent to the equal-sign form except that it creates no **typeflag g** global extended header records.) When used in **read** or **list** mode, these keyword/value pairs shall act as if they were included as records at the end of each extended header; thus, they shall override any global or file-specific extended header record keywords of the same names. For example, in the command:

```
pax -r -o "
gname:=mygroup,
" <archive
```

the group name will be forced to a new value for all files read from the archive.

The precedence of **-o** keywords over various fields in the archive is described in pax Extended Header Keyword Precedence .

-p *string*

Specify one or more file characteristic options (privileges). The *string* option-argument shall be a string specifying file characteristics to be retained or discarded on extraction. The string shall consist of the specification characters **a**, **e**, **m**, **o**, and **p** . Other implementation-defined characters can be included. Multiple characteristics can be concatenated within the same string and multiple **-p** options can be specified. The meaning of the specification characters are as follows:

a

Do not preserve file access times.

e

Preserve the user ID, group ID, file mode bits (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.168, File Mode Bits), access time, modification time, and any other implementation-defined file characteristics.

m

Do not preserve file modification times.

o

Preserve the user ID and group ID.

p

Preserve the file mode bits. Other implementation-defined file mode attributes may be preserved.

In the preceding list, "preserve" indicates that an attribute stored in the archive shall be given to the extracted file, subject to the permissions of the invoking process. The access and modification times of the file shall be preserved unless otherwise specified with the **-p** option or not stored in the archive. All attributes that are not preserved shall be determined as part of the normal file creation action (see *File Read, Write, and Creation*).

If neither the **e** nor the **o** specification character is specified, or the user ID and group ID are not preserved for any reason, *pax* shall not set the S_ISUID and S_ISGID bits of the file mode.

If the preservation of any of these items fails for any reason, *pax* shall write a diagnostic message to standard error. Failure to preserve these items shall affect the final exit status, but shall not cause the extracted file to be deleted.

If file characteristic letters in any of the *string* option-arguments are duplicated or conflict with each other, the ones given last shall take precedence. For example, if **-p eme** is specified, file modification times are preserved.

-s *replstr*

Modify file or archive member names named by *pattern* or *file* operands according to the substitution expression *replstr*, using the syntax of the *ed* utility. The concepts of "address" and "line" are meaningless in the context of the *pax* utility, and shall not be supplied. The format shall be:

-s */old/new/***[gp]**

where as in *ed*, *old* is a basic regular expression and *new* can contain an ampersand, **'\n'** (where *n* is a digit) backreferences, or subexpression matching. The *old* string shall also be permitted to contain <newline>s.

Any non-null character can be used as a delimiter (**'/'** shown here). Multiple **-s** expressions can be specified; the expressions shall be applied in the order specified, terminating with the first successful substitution. The optional trailing **'g'** is as defined in the *ed* utility. The optional trailing **'p'** shall cause successful substitutions to be written to standard error. File or archive member names that substitute to the empty string shall be ignored when reading and writing archives.

-t When reading files from the file system, and if the user has the permissions required by *utime*() to do so, set the access time of each file read to the access time that it had before being read by *pax*.

-u Ignore files that are older (having a less recent file modification time) than a pre-existing file or archive member with the same name. In **read** mode, an archive member with the same name as a file in the file system shall be extracted if the archive member is newer than the file. In **write** mode, an archive file member with the same name as a file in the file system shall be superseded if the file is newer than the archive member. If **-a** is also specified, this is accomplished by appending to the archive; otherwise, it is unspecified whether this is accomplished by actual replacement in the archive or by appending to the archive. In **copy** mode, the file in the destination hierarchy shall be replaced by the file in the source hierarchy or by a link to the file in the source hierarchy if the file in the source hierarchy is newer.

-v In **list** mode, produce a verbose table of contents (see the STDOUT section). Otherwise, write archive member pathnames to standard error (see the STDERR section).

-x *format*

Specify the output archive format. The *pax* utility shall support the following formats:

cpio

The **cpio** interchange format; see the EXTENDED DESCRIPTION section. The default *blocksize* for this format for character special archive files shall be 5120. Implementations shall support all *blocksize* values less than or equal to 32256 that are multiples of 512.

pax

The **pax** interchange format; see the EXTENDED DESCRIPTION section. The default *blocksize* for this format for character special archive files shall be 5120. Implementations shall support all *blocksize* values less than or equal to 32256 that are multiples of 512.

ustar

The **tar** interchange format; see the EXTENDED DESCRIPTION section. The default *blocksize* for this format for character special archive files shall be 10240. Implementations shall support all *blocksize* values less than or equal to 32256 that are multiples of 512.

Implementation-defined formats shall specify a default block size as well as any other block sizes supported for character special archive files.

Any attempt to append to an archive file in a format different from the existing archive format shall

cause *pax* to exit immediately with a non-zero exit status.

In **copy** mode, if no **-x** format is specified, *pax* shall behave as if **-x** *pax* were specified.

-X When traversing the file hierarchy specified by a pathname, *pax* shall not descend into directories that have a different device ID (*st_dev*; see the System Interfaces volume of IEEE Std 1003.1-2001, *stat*()).

The options that operate on the names of files or archive members (**-c**, **-i**, **-n**, **-s**, **-u**, and **-v**) shall interact as follows. In **read** mode, the archive members shall be selected based on the user-specified *pattern* operands as modified by the **-c**, **-n**, and **-u** options. Then, any **-s** and **-i** options shall modify, in that order, the names of the selected files. The **-v** option shall write names resulting from these modifications.

In **write** mode, the files shall be selected based on the user-specified pathnames as modified by the **-n** and **-u** options. Then, any **-s** and **-i** options shall modify, in that order, the names of these selected files. The **-v** option shall write names resulting from these modifications.

If both the **-u** and **-n** options are specified, *pax* shall not consider a file selected unless it is newer than the file to which it is compared.

List Mode Format Specifications

In **list** mode with the **-o listopt=** *format* option, the *format* argument shall be applied for each selected file. The *pax* utility shall append a <newline> to the **listopt** output for each selected file. The *format* argument shall be used as the *format* string described in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 5, File Format Notation, with the exceptions 1. through 5. defined in the EXTENDED DESCRIPTION section of *printf*, plus the following exceptions:

6. The sequence (*keyword*) can occur before a format conversion specifier. The conversion argument is defined by the value of *keyword*. The implementation shall support the following keywords:

   * Any of the Field Name entries in ustar Header Block and Octet-Oriented cpio Archive Entry . The implementation may support the *cpio* keywords without the leading **c**_ in addition to the form required by Values for cpio c_mode Field .

   * Any keyword defined for the extended header in pax Extended Header .

   * Any keyword provided as an implementation-defined extension within the extended header defined in pax Extended Header .

   For example, the sequence **"%(charset)s"** is the string value of the name of the character set in the extended header.

   The result of the keyword conversion argument shall be the value from the applicable header field or extended header, without any trailing NULs.

   All keyword values used as conversion arguments shall be translated from the UTF-8 encoding to the character set appropriate for the local file system, user database, and so on, as applicable.

7. An additional conversion specifier character, **T**, shall be used to specify time formats. The **T** conversion specifier character can be preceded by the sequence (*keyword*= *subformat*), where *subformat* is a date format as defined by *date* operands. The default *keyword* shall be **mtime** and the default subformat shall be:

   %b %e %H:%M %Y

8. An additional conversion specifier character, **M**, shall be used to specify the file mode string as defined in *ls* Standard Output. If (*keyword*) is omitted, the **mode** keyword shall be used. For example, **%.1M** writes the single character corresponding to the <*entry type*> field of the *ls* **-l** command.

9. An additional conversion specifier character, **D**, shall be used to specify the device for block or special files, if applicable, in an implementation-defined format. If not applicable, and (*keyword*) is specified, then this conversion shall be equivalent to **%(***keyword***)u**. If not applicable, and (*keyword*) is omitted, then this conversion shall be equivalent to <space>.

10. An additional conversion specifier character, **F**, shall be used to specify a pathname. The **F** conversion character can be preceded by a sequence of comma-separated keywords:

    (*keyword***[,***keyword***] ...)**

The values for all the keywords that are non-null shall be concatenated together, each separated by a **'/'** . The default shall be (**path**) if the keyword **path** is defined; otherwise, the default shall be (**prefix**, **name**).

11. An additional conversion specifier character, **L**, shall be used to specify a symbolic line expansion. If the current file is a symbolic link, then **%L** shall expand to:

    "%s -> %s", <*value of keyword***>, <***contents of link***>**

Otherwise, the **%L** conversion specification shall be the equivalent of **%F** .

OPERANDS

The following operands shall be supported:

directory
: The destination directory pathname for **copy** mode.

file
: A pathname of a file to be copied or archived.

pattern
: A pattern matching one or more pathnames of archive members. A pattern must be given in the name-generating notation of the pattern matching notation in *Pattern Matching Notation*, including the filename expansion rules in *Patterns Used for Filename Expansion* . The default, if no *pattern* is specified, is to select all members in the archive.

STDIN

In **write** mode, the standard input shall be used only if no *file* operands are specified. It shall be a text file containing a list of pathnames, one per line, without leading or trailing <blank>s.

In **list** and **read** modes, if **-f** is not specified, the standard input shall be an archive file.

Otherwise, the standard input shall not be used.

INPUT FILES

The input file named by the *archive* option-argument, or standard input when the archive is read from there, shall be a file formatted according to one of the specifications in the EXTENDED DESCRIPTION section or some other implementation-defined format.

The file **/dev/tty** shall be used to write prompts and read responses.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of *pax*:

LANG
: Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, Internationalization Variables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalization variables.

LC_COLLATE
: Determine the locale for the behavior of ranges, equivalence classes, and multi-character collating elements used in the pattern matching expressions for the *pattern* operand, the basic regular expression for the **-s** option, and the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for

example, single-byte as opposed to multi-byte characters in arguments and input files), the behavior of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the *LC_MESSAGES* category, and pattern matching.

LC_MESSAGES
Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the format and contents of date and time strings when the **-v** option is specified.

NLSPATH
Determine the location of message catalogs for the processing of *LC_MESSAGES* .

TMPDIR
Determine the pathname that provides part of the default global extended header record file, as described for the **-o globexthdr=** keyword in the OPTIONS section.

TZ Determine the timezone used to calculate date and time strings when the **-v** option is specified. If *TZ* is unset or null, an unspecified default timezone shall be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

In **write** mode, if **-f** is not specified, the standard output shall be the archive formatted according to one of the specifications in the EXTENDED DESCRIPTION section, or some other implementation-defined format (see **-x** *format*).

In **list** mode, when the **-o listopt=** *format* has been specified, the selected archive members shall be written to standard output using the format described under List Mode Format Specifications . In **list** mode without the **-o listopt=** *format* option, the table of contents of the selected archive members shall be written to standard output using the following format:

"%s\n", <*pathname***>**

If the **-v** option is specified in **list** mode, the table of contents of the selected archive members shall be written to standard output using the following formats.

For pathnames representing hard links to previous members of the archive:

"%s == %s\n", <*ls* **-l** *listing***>, <***linkname***>**

For all other pathnames:

"%s\n", <*ls* **-l** *listing***>**

where <*ls* -l *listing*> shall be the format specified by the *ls* utility with the **-l** option. When writing pathnames in this format, it is unspecified what is written for fields for which the underlying archive format does not have the correct information, although the correct number of <blank>-separated fields shall be written.

In **list** mode, standard output shall not be buffered more than a line at a time.

STDERR

If **-v** is specified in **read**, **write**, or **copy** modes, *pax* shall write the pathnames it processes to the standard error output using the following format:

"%s\n", <*pathname***>**

These pathnames shall be written as soon as processing is begun on the file or archive member, and shall be flushed to standard error. The trailing <newline>, which shall not be buffered, is written when the file has been read or written.

If the **-s** option is specified, and the replacement string has a trailing **'p'**, substitutions shall be written to standard error in the following format:

"%s >> %s\n", <*original pathname***>, <***new pathname***>**

In all operating modes of *pax*, optional messages of unspecified format concerning the input archive format and volume number, the number of files, blocks, volumes, and media parts as well as other diagnostic messages may be written to standard error.

In all formats, for both standard output and standard error, it is unspecified how non-printable characters in pathnames or link names are written.

When *pax* is in **read** mode or **list** mode, using the **-x pax** archive format, and a filename, link name, owner name, or any other field in an extended header record cannot be translated from the **pax** UTF-8 codeset format to the codeset and current locale of the implementation, *pax* shall write a diagnostic message to standard error, shall process the file as described for the **-o invalid=** option, and then shall process the next file in the archive.

OUTPUT FILES

In **read** mode, the extracted output files shall be of the archived file type. In **copy** mode, the copied output files shall be the type of the file being copied. In either mode, existing files in the destination hierarchy shall be overwritten only when all permission (**-p**), modification time (**-u**), and invalid-value (**-o invalid=**) tests allow it.

In **write** mode, the output file named by the **-f** option-argument shall be a file formatted according to one of the specifications in the EXTENDED DESCRIPTION section, or some other implementation-defined format.

EXTENDED DESCRIPTION

pax Interchange Format

A *pax* archive tape or file produced in the **-x pax** format shall contain a series of blocks. The physical layout of the archive shall be identical to the **ustar** format described in ustar Interchange Format . Each file archived shall be represented by the following sequence:

* An optional header block with extended header records. This header block is of the form described in pax Header Block, with a *typeflag* value of **x** or **g**. The extended header records, described in pax Extended Header, shall be included as the data for this header block.

* A header block that describes the file. Any fields in the preceding optional extended header shall override the associated fields in this header block for this file.

* Zero or more blocks that contain the contents of the file.

At the end of the archive file there shall be two 512-byte blocks filled with binary zeros, interpreted as an end-of-archive indicator.

A schematic of an example archive with global extended header records and two actual files is shown in pax Format Archive Example . In the example, the second file in the archive has no extended header preceding it, presumably because it has no need for extended attributes.

Figure: pax Format Archive Example

pax Header Block

The **pax** header block shall be identical to the **ustar** header block described in ustar Interchange Format, except that two additional *typeflag* values are defined:

x Represents extended header records for the following file in the archive (which shall have its own **ustar** header block). The format of these extended header records shall be as described in pax Extended Header .

g Represents global extended header records for the following files in the archive. The format of these extended header records shall be as described in pax Extended Header . Each value shall affect all subsequent files that do not override that value in their own extended header record and until another global extended header record is reached that provides another value for the same field. The *typeflag* **g** global headers should not be used with interchange media that could suffer partial data loss in transporting the archive.

For both of these types, the *size* field shall be the size of the extended header records in octets. The other fields in the header block are not meaningful to this version of the *pax* utility. However, if this archive is read by a *pax* utility conforming to the ISO POSIX-2:1993 standard, the header block fields

are used to create a regular file that contains the extended header records as data. Therefore, header block field values should be selected to provide reasonable file access to this regular file.

A further difference from the **ustar** header block is that data blocks for files of *typeflag* 1 (the digit one) (hard link) may be included, which means that the size field may be greater than zero. Archives created by *pax* **-o linkdata** shall include these data blocks with the hard links.

pax Extended Header

A **pax** extended header contains values that are inappropriate for the **ustar** header block because of limitations in that format: fields requiring a character encoding other than that described in the ISO/IEC 646:1991 standard, fields representing file attributes not described in the **ustar** header, and fields whose format or length do not fit the requirements of the **ustar** header. The values in an extended header add attributes to the following file (or files; see the description of the *typeflag* **g** header block) or override values in the following header block(s), as indicated in the following list of keywords.

An extended header shall consist of one or more records, each constructed as follows:

"%d %s=%s\n", <*length***>, <***keyword***>, <***value***>**

The extended header records shall be encoded according to the ISO/IEC 10646-1:2000 standard (UTF-8). The <*length*> field, <blank>, equals sign, and <newline> shown shall be limited to the portable character set, as encoded in UTF-8. The <*keyword*> and <*value*> fields can be any UTF-8 characters. The <*length*> field shall be the decimal length of the extended header record in octets, including the trailing <newline>.

The <*keyword*> field shall be one of the entries from the following list or a keyword provided as an implementation extension. Keywords consisting entirely of lowercase letters, digits, and periods are reserved for future standardization. A keyword shall not include an equals sign. (In the following list, the notations "file(s)" or "block(s)" is used to acknowledge that a keyword affects the following single file after a *typeflag* **x** extended header, but possibly multiple files after *typeflag* **g**. Any requirements in the list for *pax* to include a record when in **write** or **copy** mode shall apply only when such a record has not already been provided through the use of the **-o** option. When used in **copy** mode, *pax* shall behave as if an archive had been created with applicable extended header records and then extracted.)

atime The file access time for the following file(s), equivalent to the value of the *st_atime* member of the **stat** structure for a file, as described by the *stat*() function. The access time shall be restored if the process has the appropriate privilege required to do so. The format of the <*value*> shall be as described in pax Extended Header File Times .

charset

The name of the character set used to encode the data in the following file(s). The entries in the following table are defined to refer to known standards; additional names may be agreed on between the originator and recipient.

<value>	**Formal Standard**
ISO-IR 646 1990	ISO/IEC 646:1990
ISO-IR 8859 1 1998	ISO/IEC 8859-1:1998
ISO-IR 8859 2 1999	ISO/IEC 8859-2:1999
ISO-IR 8859 3 1999	ISO/IEC 8859-3:1999
ISO-IR 8859 4 1998	ISO/IEC 8859-4:1998
ISO-IR 8859 5 1999	ISO/IEC 8859-5:1999
ISO-IR 8859 6 1999	ISO/IEC 8859-6:1999
ISO-IR 8859 7 1987	ISO/IEC 8859-7:1987
ISO-IR 8859 8 1999	ISO/IEC 8859-8:1999
ISO-IR 8859 9 1999	ISO/IEC 8859-9:1999
ISO-IR 8859 10 1998	ISO/IEC 8859-10:1998
ISO-IR 8859 13 1998	ISO/IEC 8859-13:1998
ISO-IR 8859 14 1998	ISO/IEC 8859-14:1998
ISO-IR 8859 15 1999	ISO/IEC 8859-15:1999
ISO-IR 10646 2000	ISO/IEC 10646:2000
ISO-IR 10646 2000 UTF-8	ISO/IEC 10646, UTF-8 encoding
BINARY	None.

The encoding is included in an extended header for information only; when *pax* is used as described in

IEEE Std 1003.1-2001, it shall not translate the file data into any other encoding. The **BINARY** entry indicates unencoded binary data.

When used in **write** or **copy** mode, it is implementation-defined whether *pax* includes a **charset** extended header record for a file.

comment
A series of characters used as a comment. All characters in the <*value*> field shall be ignored by *pax*.

ctime The file creation time for the following file(s), equivalent to the value of the *st_ctime* member of the **stat** structure for a file, as described by the *stat*() function. The creation time shall be restored if the process has the appropriate privilege required to do so. The format of the <*value*> shall be as described in pax Extended Header File Times .

gid The group ID of the group that owns the file, expressed as a decimal number using digits from the ISO/IEC 646:1991 standard. This record shall override the *gid* field in the following header block(s). When used in **write** or **copy** mode, *pax* shall include a *gid* extended header record for each file whose group ID is greater than 2097151 (octal 7777777).

gname The group of the file(s), formatted as a group name in the group database. This record shall override the *gid* and *gname* fields in the following header block(s), and any *gid* extended header record. When used in **read**, **copy**, or **list** mode, *pax* shall translate the name from the UTF-8 encoding in the header record to the character set appropriate for the group database on the receiving system. If any of the UTF-8 characters cannot be translated, and if the **-o invalid=** UTF-8 option is not specified, the results are implementation-defined. When used in **write** or **copy** mode, *pax* shall include a **gname** extended header record for each file whose group name cannot be represented entirely with the letters and digits of the portable character set.

linkpath
The pathname of a link being created to another file, of any type, previously archived. This record shall override the *linkname* field in the following **ustar** header block(s). The following **ustar** header block shall determine the type of link created. If *typeflag* of the following header block is 1, it shall be a hard link. If *typeflag* is 2, it shall be a symbolic link and the **linkpath** value shall be the contents of the symbolic link. The *pax* utility shall translate the name of the link (contents of the symbolic link) from the UTF-8 encoding to the character set appropriate for the local file system. When used in **write** or **copy** mode, *pax* shall include a **linkpath** extended header record for each link whose pathname cannot be represented entirely with the members of the portable character set other than NUL.

mtime The file modification time of the following file(s), equivalent to the value of the *st_mtime* member of the **stat** structure for a file, as described in the *stat*() function. This record shall override the *mtime* field in the following header block(s). The modification time shall be restored if the process has the appropriate privilege required to do so. The format of the <*value*> shall be as described in pax Extended Header File Times .

path The pathname of the following file(s). This record shall override the *name* and *prefix* fields in the following header block(s). The *pax* utility shall translate the pathname of the file from the UTF-8 encoding to the character set appropriate for the local file system.

When used in **write** or **copy** mode, *pax* shall include a *path* extended header record for each file whose pathname cannot be represented entirely with the members of the portable character set other than NUL.

realtime.*any*
The keywords prefixed by "realtime." are reserved for future standardization.

security.*any*
The keywords prefixed by "security." are reserved for future standardization.

size The size of the file in octets, expressed as a decimal number using digits from the ISO/IEC 646:1991 standard. This record shall override the *size* field in the following header block(s). When used in **write** or **copy** mode, *pax* shall include a *size* extended header record for each file with a size value greater than 8589934591 (octal 77777777777).

uid The user ID of the file owner, expressed as a decimal number using digits from the ISO/IEC 646:1991 standard. This record shall override the *uid* field in the following header block(s). When used in **write** or **copy** mode, *pax* shall include a *uid* extended header record for each file whose owner ID is greater than 2097151 (octal 7777777).

uname The owner of the following file(s), formatted as a user name in the user database. This record shall override the *uid* and *uname* fields in the following header block(s), and any *uid* extended header record. When used in **read**, **copy**, or **list** mode, *pax* shall translate the name from the UTF-8 encoding in the header record to the character set appropriate for the user database on the receiving system. If any of the UTF-8 characters cannot be translated, and if the **-o invalid=** UTF-8 option is not specified, the results are implementation-defined. When used in **write** or **copy** mode, *pax* shall include a **uname** extended header record for each file whose user name cannot be represented entirely with the letters and digits of the portable character set.

If the <*value*> field is zero length, it shall delete any header block field, previously entered extended header value, or global extended header value of the same name.

If a keyword in an extended header record (or in a **-o** option-argument) overrides or deletes a corresponding field in the **ustar** header block, *pax* shall ignore the contents of that header block field.

Unlike the **ustar** header block fields, NULs shall not delimit <*value*>s; all characters within the <*value*> field shall be considered data for the field. None of the length limitations of the **ustar** header block fields in ustar Header Block shall apply to the extended header records.

pax Extended Header Keyword Precedence

This section describes the precedence in which the various header records and fields and command line options are selected to apply to a file in the archive. When *pax* is used in **read** or **list** modes, it shall determine a file attribute in the following sequence:

1. If **-o delete=** *keyword-prefix* is used, the affected attributes shall be determined from step 7., if applicable, or ignored otherwise.
2. If **-o** *keyword*:= is used, the affected attributes shall be ignored.
3. If **-o** *keyword* **:=** *value* is used, the affected attribute shall be assigned the value.
4. If there is a *typeflag* **x** extended header record, the affected attribute shall be assigned the <*value*>. When extended header records conflict, the last one given in the header shall take precedence.
5. If **-o** *keyword* **=** *value* is used, the affected attribute shall be assigned the value.
6. If there is a *typeflag* **g** global extended header record, the affected attribute shall be assigned the <*value*>. When global extended header records conflict, the last one given in the global header shall take precedence.
7. Otherwise, the attribute shall be determined from the **ustar** header block.

pax Extended Header File Times

The *pax* utility shall write an **mtime** record for each file in **write** or **copy** modes if the file's modification time cannot be represented exactly in the **ustar** header logical record described in ustar Interchange Format . This can occur if the time is out of **ustar** range, or if the file system of the underlying implementation supports non-integer time granularities and the time is not an integer. All of these time records shall be formatted as a decimal representation of the time in seconds since the Epoch. If a period (**'.'**) decimal point character is present, the digits to the right of the point shall represent the units of a subsecond timing granularity, where the first digit is tenths of a second and each subsequent digit is a tenth of the previous digit. In **read** or **copy** mode, the *pax* utility shall truncate the time of a file to the greatest value that is not greater than the input header file time. In **write** or **copy** mode, the *pax* utility shall output a time exactly if it can be represented exactly as a decimal number, and otherwise shall generate only enough digits so that the same time shall be recovered if the file is extracted on a system whose underlying implementation supports the same time granularity.

ustar Interchange Format

A **ustar** archive tape or file shall contain a series of logical records. Each logical record shall be a fixed-size logical record of 512 octets (see below). Although this format may be thought of as being stored on 9-track industry-standard 12.7 mm (0.5 in) magnetic tape, other types of transportable media are not excluded. Each file archived shall be represented by a header logical record that describes the file, followed by zero or more logical records that give the contents of the file. At the end of the archive file there shall be two 512-octet logical records filled with binary zeros, interpreted as an end-of-archive indicator.

The logical records may be grouped for physical I/O operations, as described under the **-b** *blocksize* and **-x ustar** options. Each group of logical records may be written with a single operation equivalent to the *write*() function. On magnetic tape, the result of this write shall be a single tape physical block. The last physical block shall always be the full size, so logical records after the two zero logical records may contain undefined data.

The header logical record shall be structured as shown in the following table. All lengths and offsets are in decimal.

Table: ustar Header Block

Field Name	Octet Offset	Length (in Octets)
name	0	100
mode	100	8
uid	108	8
gid	116	8
size	124	12
mtime	136	12
chksum	148	8
typeflag	156	1
linkname	157	100
magic	257	6
version	263	2
uname	265	32
gname	297	32
devmajor	329	8
devminor	337	8
prefix	345	155

All characters in the header logical record shall be represented in the coded character set of the ISO/IEC 646:1991 standard. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as octets with the most significant bit zero. If an implementation supports the use of characters outside of slash and the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes.

However, the *pax* utility shall never create filenames on the local system that cannot be accessed via the procedures described in IEEE Std 1003.1-2001. If a filename is found on the medium that would create an invalid filename, it is implementation-defined whether the data from the file is stored on the file hierarchy and under what name it is stored. The *pax* utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

Each field within the header logical record is contiguous; that is, there is no padding used. Each character on the archive medium shall be stored contiguously.

The fields *magic*, *uname*, and *gname* are character strings each terminated by a NUL character. The fields *name*, *linkname*, and *prefix* are NUL-terminated character strings except when all characters in the array contain non-NUL characters including the last character. The *version* field is two octets containing the characters **"00"** (zero-zero). The *typeflag* contains a single character. All other fields are leading zero-filled octal numbers using digits from the ISO/IEC 646:1991 standard IRV. Each numeric field is terminated by one or more <space> or NUL characters.

The *name* and the *prefix* fields shall produce the pathname of the file. A new pathname shall be formed, if *prefix* is not an empty string (its first character is not NUL), by concatenating *prefix* (up to the first NUL character), a slash character, and *name*; otherwise, *name* is used alone. In either case, *name* is

terminated at the first NUL character. If *prefix* begins with a NUL character, it shall be ignored. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, *pax* shall notify the user of the error, and shall not store any part of the file-header or data-on the medium.

The *linkname* field, described below, shall not use the *prefix* to produce a pathname. As such, a *linkname* is limited to 100 characters. If the name does not fit in the space provided, *pax* shall notify the user of the error, and shall not attempt to store the link on the medium.

The *mode* field provides 12 bits encoded in the ISO/IEC 646:1991 standard octal digit representation. The encoded bits shall represent the following values:

Table: ustar *mode* Field

Bit Value	IEEE Std 1003.1-2001 Bit	Description
04000	*S_ISUID*	*Set UID on execution.*
02000	*S_ISGID*	*Set GID on execution.*
01000	*<reserved>*	*Reserved for future standardization.*
00400	*S_IRUSR*	*Read permission for file owner class.*
00200	*S_IWUSR*	*Write permission for file owner class.*
00100	*S_IXUSR*	*Execute/search permission for file owner class.*
00040	*S_IRGRP*	*Read permission for file group class.*
00020	*S_IWGRP*	*Write permission for file group class.*
00010	*S_IXGRP*	*Execute/search permission for file group class.*
00004	*S_IROTH*	*Read permission for file other class.*
00002	*S_IWOTH*	*Write permission for file other class.*
00001	*S_IXOTH*	*Execute/search permission for file other class.*

When appropriate privilege is required to set one of these mode bits, and the user restoring the files from the archive does not have the appropriate privilege, the mode bits for which the user does not have appropriate privilege shall be ignored. Some of the mode bits in the archive format are not mentioned elsewhere in this volume of IEEE Std 1003.1-2001. If the implementation does not support those bits, they may be ignored.

The *uid* and *gid* fields are the user and group ID of the owner and group of the file, respectively.

The *size* field is the size of the file in octets. If the *typeflag* field is set to specify a file to be of type 1 (a link) or 2 (a symbolic link), the *size* field shall be specified as zero. If the *typeflag* field is set to specify a file of type 5 (directory), the *size* field shall be interpreted as described under the definition of that record type. No data logical records are stored for types 1, 2, or 5. If the *typeflag* field is set to 3 (character special file), 4 (block special file), or 6 (FIFO), the meaning of the *size* field is unspecified by this volume of IEEE Std 1003.1-2001, and no data logical records shall be stored on the medium. Additionally, for type 6, the *size* field shall be ignored when reading. If the *typeflag* field is set to any other value, the number of logical records written following the header shall be (*size*+511)/512, ignoring any fraction in the result of the division.

The *mtime* field shall be the modification time of the file at the time it was archived. It is the ISO/IEC 646:1991 standard representation of the octal value of the modification time obtained from the *stat*() function.

The *chksum* field shall be the ISO/IEC 646:1991 standard IRV representation of the octal value of the simple sum of all octets in the header logical record. Each octet in the header shall be treated as an unsigned value. These values shall be added to an unsigned integer, initialized to zero, the precision of which is not less than 17 bits. When calculating the checksum, the *chksum* field is treated as if it were all spaces.

The *typeflag* field specifies the type of file archived. If a particular implementation does not recognize the type, or the user does not have appropriate privilege to create that type, the file shall be extracted as if it were a regular file if the file type is defined to have a meaning for the *size* field that could cause data logical records to be written on the medium (see the previous description for *size*). If conversion to a regular file occurs, the *pax* utility shall produce an error indicating that the conversion took place. All of the *typeflag* fields shall be coded in the ISO/IEC 646:1991 standard IRV:

0 Represents a regular file. For backwards-compatibility, a *typeflag* value of binary zero (**'\0'**) should be recognized as meaning a regular file when extracting files from the archive. Archives written with this version of the archive file format create regular files with a *typeflag* value of the ISO/IEC 646:1991 standard IRV **'0'** .

1 Represents a file linked to another file, of any type, previously archived. Such files are identified by each file having the same device and file serial number. The linked-to name is specified in the *linkname* field with a NUL-character terminator if it is less than 100 octets in length.

2 Represents a symbolic link. The contents of the symbolic link shall be stored in the *linkname* field.

3,4 Represent character special files and block special files respectively. In this case the *devmajor* and *devminor* fields shall contain information defining the device, the format of which is unspecified by this volume of IEEE Std 1003.1-2001. Implementations may map the device specifications to their own local specification or may ignore the entry.

5 Specifies a directory or subdirectory. On systems where disk allocation is performed on a directory basis, the *size* field shall contain the maximum number of octets (which may be rounded to the nearest disk block allocation unit) that the directory may hold. A *size* field of zero indicates no such limiting. Systems that do not support limiting in this manner should ignore the *size* field.

6 Specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.

7 Reserved to represent a file to which an implementation has associated some high-performance attribute. Implementations without such extensions should treat this file as a regular file (type 0).

A-Z The letters **'A'** to **'Z'**, inclusive, are reserved for custom implementations. All other values are reserved for future versions of IEEE Std 1003.1-2001.

Attempts to archive a socket using **ustar** interchange format shall produce a diagnostic message. Handling of other file types is implementation-defined.

The *magic* field is the specification that this archive was output in this archive format. If this field contains **ustar** (the five characters from the ISO/IEC 646:1991 standard IRV shown followed by NUL), the *uname* and *gname* fields shall contain the ISO/IEC 646:1991 standard IRV representation of the owner and group of the file, respectively (truncated to fit, if necessary). When the file is restored by a privileged, protection-preserving version of the utility, the user and group databases shall be scanned for these names. If found, the user and group IDs contained within these files shall be used rather than the values contained within the *uid* and *gid* fields.

cpio Interchange Format

The octet-oriented **cpio** archive format shall be a series of entries, each comprising a header that describes the file, the name of the file, and then the contents of the file.

An archive may be recorded as a series of fixed-size blocks of octets. This blocking shall be used only to make physical I/O more efficient. The last group of blocks shall always be at the full size.

For the octet-oriented **cpio** archive format, the individual entry information shall be in the order indicated and described by the following table; see also the <*cpio.h*> header.

Table: Octet-Oriented cpio Archive Entry

Header Field Name	Length (in Octets)	Interpreted as
c_magic	6	Octal number
c_dev	6	Octal number
c_ino	6	Octal number
c_mode	6	Octal number
c_uid	6	Octal number
c_gid	6	Octal number
c_nlink	6	Octal number

c_rdev	6	Octal number
c_mtime	11	Octal number
c_namesize	6	Octal number
c_filesize	11	Octal number
Filename Field Name	**Length**	**Interpreted as**
c_name	*c_namesize*	Pathname string
File Data Field Name	**Length**	**Interpreted as**
c_filedata	*c_filesize*	Data

cpio Header

For each file in the archive, a header as defined previously shall be written. The information in the header fields is written as streams of the ISO/IEC 646:1991 standard characters interpreted as octal numbers. The octal numbers shall be extended to the necessary length by appending the ISO/IEC 646:1991 standard IRV zeros at the most-significant-digit end of the number; the result is written to the most-significant digit of the stream of octets first. The fields shall be interpreted as follows:

c_magic

Identify the archive as being a transportable archive by containing the identifying value **"070707"** .

c_dev, *c_ino*

Contains values that uniquely identify the file within the archive (that is, no files contain the same pair of *c_dev* and *c_ino* values unless they are links to the same file). The values shall be determined in an unspecified manner.

c_mode

Contains the file type and access permissions as defined in the following table.

Table: Values for cpio c_mode Field

File Permissions Name	**Value**	**Indicates**
C_IRUSR	000400	Read by owner
C_IWUSR	000200	Write by owner
C_IXUSR	000100	Execute by owner
C_IRGRP	000040	Read by group
C_IWGRP	000020	Write by group
C_IXGRP	000010	Execute by group
C_IROTH	000004	Read by others
C_IWOTH	000002	Write by others
C_IXOTH	000001	Execute by others
C_ISUID	004000	Set *uid*
C_ISGID	002000	Set *gid*
C_ISVTX	001000	Reserved
File Type Name	**Value**	**Indicates**
C_ISDIR	040000	Directory
C_ISFIFO	010000	FIFO
C_ISREG	0100000	Regular file
C_ISLNK	0120000	Symbolic link
C_ISBLK	060000	Block special file
C_ISCHR	020000	Character special file
C_ISSOCK	0140000	Socket
C_ISCTG	0110000	Reserved

Directories, FIFOs, symbolic links, and regular files shall be supported on a system conforming to this volume of IEEE Std 1003.1-2001; additional values defined previously are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written to archives intended to be transported to other systems.

c_uid Contains the user ID of the owner.

c_gid Contains the group ID of the group.

c_nlink Contains the number of links referencing the file at the time the archive was created.

c_rdev Contains implementation-defined information for character or block special files.

c_mtime
Contains the latest time of modification of the file at the time the archive was created.

c_namesize
Contains the length of the pathname, including the terminating NUL character.

c_filesize
Contains the length of the file in octets. This shall be the length of the data section following the header structure.

cpio Filename

The *c_name* field shall contain the pathname of the file. The length of this field in octets is the value of *c_namesize*.

If a filename is found on the medium that would create an invalid pathname, it is implementation-defined whether the data from the file is stored on the file hierarchy and under what name it is stored.

All characters shall be represented in the ISO/IEC 646:1991 standard IRV. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as octets with the most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users, and groups, one or more implementation-defined encodings of these characters shall be provided for interchange purposes. However, the *pax* utility shall never create filenames on the local system that cannot be accessed via the procedures described previously in this volume of IEEE Std 1003.1-2001. If a filename is found on the medium that would create an invalid filename, it is implementation-defined whether the data from the file is stored on the local file system and under what name it is stored. The *pax* utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

cpio File Data

Following *c_name*, there shall be *c_filesize* octets of data. Interpretation of such data occurs in a manner dependent on the file. If *c_filesize* is zero, no data shall be contained in *c_filedata*.

When restoring from an archive:

* If the user does not have the appropriate privilege to create a file of the specified type, *pax* shall ignore the entry and write an error message to standard error.

* Only regular files have data to be restored. Presuming a regular file meets any selection criteria that might be imposed on the format-reading utility by the user, such data shall be restored.

* If a user does not have appropriate privilege to set a particular mode flag, the flag shall be ignored. Some of the mode flags in the archive format are not mentioned elsewhere in this volume of IEEE Std 1003.1-2001. If the implementation does not support those flags, they may be ignored.

cpio Special Entries

FIFO special files, directories, and the trailer shall be recorded with *c_filesize* equal to zero. For other special files, *c_filesize* is unspecified by this volume of IEEE Std 1003.1-2001. The header for the next file entry in the archive shall be written directly after the last octet of the file entry preceding it. A header denoting the filename **TRAILER!!!** shall indicate the end of the archive; the contents of octets in the last block of the archive following such a header are undefined.

EXIT STATUS

The following exit values shall be returned:

0 All files were processed successfully.

>0 An error occurred.

CONSEQUENCES OF ERRORS

If *pax* cannot create a file or a link when reading an archive or cannot find a file when writing an

archive, or cannot preserve the user ID, group ID, or file mode when the **-p** option is specified, a diagnostic message shall be written to standard error and a non-zero exit status shall be returned, but processing shall continue. In the case where *pax* cannot create a link to a file, *pax* shall not, by default, create a second copy of the file.

If the extraction of a file from an archive is prematurely terminated by a signal or error, *pax* may have only partially extracted the file or (if the **-n** option was not specified) may have extracted a file of the same name as that specified by the user, but which is not the file the user wanted. Additionally, the file modes of extracted directories may have additional bits from the S_IRWXU mask set as well as incorrect modification and access times.

The following sections are informative.

APPLICATION USAGE

The **-p** (privileges) option was invented to reconcile differences between historical *tar* and *cpio* implementations. In particular, the two utilities use **-m** in diametrically opposed ways. The **-p** option also provides a consistent means of extending the ways in which future file attributes can be addressed, such as for enhanced security systems or high-performance files. Although it may seem complex, there are really two modes that are most commonly used:

-p e "Preserve everything". This would be used by the historical superuser, someone with all the appropriate privileges, to preserve all aspects of the files as they are recorded in the archive. The **e** flag is the sum of **o** and **p**, and other implementation-defined attributes.

-p p "Preserve" the file mode bits. This would be used by the user with regular privileges who wished to preserve aspects of the file other than the ownership. The file times are preserved by default, but two other flags are offered to disable these and use the time of extraction.

The one pathname per line format of standard input precludes pathnames containing <newline>s. Although such pathnames violate the portable filename guidelines, they may exist and their presence may inhibit usage of *pax* within shell scripts. This problem is inherited from historical archive programs. The problem can be avoided by listing filename arguments on the command line instead of on standard input.

It is almost certain that appropriate privileges are required for *pax* to accomplish parts of this volume of IEEE Std 1003.1-2001. Specifically, creating files of type block special or character special, restoring file access times unless the files are owned by the user (the **-t** option), or preserving file owner, group, and mode (the **-p** option) all probably require appropriate privileges.

In **read** mode, implementations are permitted to overwrite files when the archive has multiple members with the same name. This may fail if permissions on the first version of the file do not permit it to be overwritten.

The **cpio** and **ustar** formats can only support files up to 8589934592 bytes (8 * 2^30) in size.

EXAMPLES

The following command:

```
pax -w -f /dev/rmt/1m .
```

copies the contents of the current directory to tape drive 1, medium density (assuming historical System V device naming procedures-the historical BSD device name would be **/dev/rmt9**).

The following commands:

```
mkdir newdirpax -rw olddir newdir
```

copy the *olddir* directory hierarchy to *newdir*.

```
pax -r -s ',^//*usr//*,,' -f a.pax
```

reads the archive **a.pax**, with all files rooted in **/usr** in the archive extracted relative to the current directory.

Using the option:

```
-o listopt="%M %(atime)T %(size)D %(name)s"
```

overrides the default output description in Standard Output and instead writes:

```
-rw-rw--- Jan 12 15:53 1492 /usr/foo/bar
```

Using the options:

```
-o listopt='%L\t%(size)D\n%.7' \
-o listopt='(name)s\n%(ctime)T\n%T'
```

overrides the default output description in Standard Output and instead writes:

```
/usr/foo/bar -> /tmp   1492
/usr/fo
Jan 12 1991
Jan 31 15:53
```

RATIONALE

The *pax* utility was new for the ISO POSIX-2:1993 standard. It represents a peaceful compromise between advocates of the historical *tar* and *cpio* utilities.

A fundamental difference between *cpio* and *tar* was in the way directories were treated. The *cpio* utility did not treat directories differently from other files, and to select a directory and its contents required that each file in the hierarchy be explicitly specified. For *tar*, a directory matched every file in the file hierarchy it rooted.

The *pax* utility offers both interfaces; by default, directories map into the file hierarchy they root. The **-d** option causes *pax* to skip any file not explicitly referenced, as *cpio* historically did. The *tar* **-** *style* behavior was chosen as the default because it was believed that this was the more common usage and because *tar* is the more commonly available interface, as it was historically provided on both System V and BSD implementations.

The data interchange format specification in this volume of IEEE Std 1003.1-2001 requires that processes with "appropriate privileges" shall always restore the ownership and permissions of extracted files exactly as archived. If viewed from the historic equivalence between superuser and "appropriate privileges", there are two problems with this requirement. First, users running as superusers may unknowingly set dangerous permissions on extracted files. Second, it is needlessly limiting, in that superusers cannot extract files and own them as superuser unless the archive was created by the superuser. (It should be noted that restoration of ownerships and permissions for the superuser, by default, is historical practice in *cpio*, but not in *tar*.) In order to avoid these two problems, the *pax* specification has an additional "privilege" mechanism, the **-p** option. Only a *pax* invocation with the privileges needed, and which has the **-p** option set using the **e** specification character, has the "appropriate privilege" to restore full ownership and permission information.

Note also that this volume of IEEE Std 1003.1-2001 requires that the file ownership and access permissions shall be set, on extraction, in the same fashion as the *creat*() function when provided with the mode stored in the archive. This means that the file creation mask of the user is applied to the file permissions.

Users should note that directories may be created by *pax* while extracting files with permissions that are different from those that existed at the time the archive was created. When extracting sensitive information into a directory hierarchy that no longer exists, users are encouraged to set their file creation mask appropriately to protect these files during extraction.

The table of contents output is written to standard output to facilitate pipeline processing.

An early proposal had hard links displaying for all pathnames. This was removed because it complicates the output of the case where **-v** is not specified and does not match historical *cpio* usage. The hard-link information is available in the **-v** display.

The description of the **-l** option allows implementations to make hard links to symbolic links. IEEE Std 1003.1-2001 does not specify any way to create a hard link to a symbolic link, but many implementations provide this capability as an extension. If there are hard links to symbolic links when an archive is created, the implementation is required to archive the hard link in the archive (unless **-H** or **-L** is specified). When in **read** mode and in **copy** mode, implementations supporting hard links to symbolic links should use them when appropriate.

The archive formats inherited from the POSIX.1-1990 standard have certain restrictions that have been brought along from historical usage. For example, there are restrictions on the length of pathnames stored in the archive. When *pax* is used in **copy**(**-rw**) mode (copying directory hierarchies), the ability to use extensions from the **-x pax** format overcomes these restrictions.

The default *blocksize* value of 5120 bytes for *cpio* was selected because it is one of the standard block-size values for *cpio*, set when the **-B** option is specified. (The other default block-size value for *cpio* is 512 bytes, and this was considered to be too small.) The default block value of 10240 bytes for *tar* was selected because that is the standard block-size value for BSD *tar*. The maximum block size of 32256 bytes (2**15-512 bytes) is the largest multiple of 512 bytes that fits into a signed 16-bit tape controller transfer register. There are known limitations in some historical systems that would prevent larger blocks from being accepted. Historical values were chosen to improve compatibility with historical scripts using *dd* or similar utilities to manipulate archives. Also, default block sizes for any file type other than character special file has been deleted from this volume of IEEE Std 1003.1-2001 as unimportant and not likely to affect the structure of the resulting archive.

Implementations are permitted to modify the block-size value based on the archive format or the device to which the archive is being written. This is to provide implementations with the opportunity to take advantage of special types of devices, and it should not be used without a great deal of consideration as it almost certainly decreases archive portability.

The intended use of the **-n** option was to permit extraction of one or more files from the archive without processing the entire archive. This was viewed by the standard developers as offering significant performance advantages over historical implementations. The **-n** option in early proposals had three effects; the first was to cause special characters in patterns to not be treated specially. The second was to cause only the first file that matched a pattern to be extracted. The third was to cause *pax* to write a diagnostic message to standard error when no file was found matching a specified pattern. Only the second behavior is retained by this volume of IEEE Std 1003.1-2001, for many reasons. First, it is in general not acceptable for a single option to have multiple effects. Second, the ability to make pattern matching characters act as normal characters is useful for parts of *pax* other than file extraction. Third, a finer degree of control over the special characters is useful because users may wish to normalize only a single special character in a single filename. Fourth, given a more general escape mechanism, the previous behavior of the **-n** option can be easily obtained using the **-s** option or a *sed* script. Finally, writing a diagnostic message when a pattern specified by the user is unmatched by any file is useful behavior in all cases.

In this version, the **-n** was removed from the **copy** mode synopsis of *pax*; it is inapplicable because there are no pattern operands specified in this mode.

There is another method than *pax* for copying subtrees in IEEE Std 1003.1-2001 described as part of the *cp* utility. Both methods are historical practice: *cp* provides a simpler, more intuitive interface, while *pax* offers a finer granularity of control. Each provides additional functionality to the other; in particular, *pax* maintains the hard-link structure of the hierarchy while *cp* does not. It is the intention of the standard developers that the results be similar (using appropriate option combinations in both utilities). The results are not required to be identical; there seemed insufficient gain to applications to balance the difficulty of implementations having to guarantee that the results would be exactly identical.

A single archive may span more than one file. It is suggested that implementations provide informative messages to the user on standard error whenever the archive file is changed.

The **-d** option (do not create intermediate directories not listed in the archive) found in early proposals was originally provided as a complement to the historic **-d** option of *cpio*. It has been deleted.

The **-s** option in early proposals specified a subset of the substitution command from the *ed* utility. As there was no reason for only a subset to be supported, the **-s** option is now compatible with the current *ed* specification. Since the delimiter can be any non-null character, the following usage with single spaces is valid:

```
pax -s " foo bar " ...
```

The **-t** description is worded so as to note that this may cause the access time update caused by some other activity (which occurs while the file is being read) to be overwritten.

The default behavior of *pax* with regard to file modification times is the same as historical implementations of *tar*. It is not the historical behavior of *cpio*.

Because the **-i** option uses **/dev/tty**, utilities without a controlling terminal are not able to use this option.

The **-y** option, found in early proposals, has been deleted because a line containing a single period for the **-i** option has equivalent functionality. The special lines for the **-i** option (a single period and the empty line) are historical practice in *cpio*.

In early drafts, a **-e** *charmap* option was included to increase portability of files between systems using different coded character sets. This option was omitted because it was apparent that consensus could not be formed for it. In this version, the use of UTF-8 should be an adequate substitute.

The **-k** option was added to address international concerns about the dangers involved in the character set transformations of **-e** (if the target character set were different from the source, the filenames might be transformed into names matching existing files) and also was made more general to protect files transferred between file systems with different {NAME_MAX} values (truncating a filename on a smaller system might also inadvertently overwrite existing files). As stated, it prevents any overwriting, even if the target file is older than the source. This version adds more granularity of options to solve this problem by introducing the **-o invalid=** option-specifically the UTF-8 action. (Note that an existing file that is named with a UTF-8 encoding is still subject to overwriting in this case. The **-k** option closes that loophole.)

Some of the file characteristics referenced in this volume of IEEE Std 1003.1-2001 might not be supported by some archive formats. For example, neither the **tar** nor **cpio** formats contain the file access time. For this reason, the **e** specification character has been provided, intended to cause all file characteristics specified in the archive to be retained.

It is required that extracted directories, by default, have their access and modification times and permissions set to the values specified in the archive. This has obvious problems in that the directories are almost certainly modified after being extracted and that directory permissions may not permit file creation. One possible solution is to create directories with the mode specified in the archive, as modified by the *umask* of the user, with sufficient permissions to allow file creation. After all files have been extracted, *pax* would then reset the access and modification times and permissions as necessary.

The list-mode formatting description borrows heavily from the one defined by the *printf* utility. However, since there is no separate operand list to get conversion arguments, the format was extended to allow specifying the name of the conversion argument as part of the conversion specification.

The **T** conversion specifier allows time fields to be displayed in any of the date formats. Unlike the *ls* utility, *pax* does not adjust the format when the date is less than six months in the past. This makes parsing the output more predictable.

The **D** conversion specifier handles the ability to display the major/minor or file size, as with *ls*, by using **%-8(***size***)D**.

The **L** conversion specifier handles the *ls* display for symbolic links.

Conversion specifiers were added to generate existing known types used for *ls*.

pax Interchange Format

The new POSIX data interchange format was developed primarily to satisfy international concerns that the **ustar** and **cpio** formats did not provide for file, user, and group names encoded in characters outside a subset of the ISO/IEC 646:1991 standard. The standard developers realized that this new POSIX data interchange format should be very extensible because there were other requirements they foresaw in the near future:

* Support international character encodings and locale information
* Support security information (ACLs, and so on)
* Support future file types, such as realtime or contiguous files
* Include data areas for implementation use
* Support systems with words larger than 32 bits and timers with subsecond granularity

The following were not goals for this format because these are better handled by separate utilities or are inappropriate for a portable format:

* Encryption
* Compression
* Data translation between locales and codesets
* *inode* storage

The format chosen to support the goals is an extension of the **ustar** format. Of the two formats previously available, only the **ustar** format was selected for extensions because:

* It was easier to extend in an upwards-compatible way. It offered version flags and header block type fields with room for future standardization. The **cpio** format, while possessing a more flexible file naming methodology, could not be extended without breaking some theoretical implementation or using a dummy filename that could be a legitimate filename.
* Industry experience since the original " *tar* wars" fought in developing the ISO POSIX-1 standard has clearly been in favor of the **ustar** format, which is generally the default output format selected for *pax* implementations on new systems.

The new format was designed with one additional goal in mind: reasonable behavior when an older *tar* or *pax* utility happened to read an archive. Since the POSIX.1-1990 standard mandated that a "format-reading utility" had to treat unrecognized *typeflag* values as regular files, this allowed the format to include all the extended information in a pseudo-regular file that preceded each real file. An option is given that allows the archive creator to set up reasonable names for these files on the older systems. Also, the normative text suggests that reasonable file access values be used for this **ustar** header block. Making these header files inaccessible for convenient reading and deleting would not be reasonable. File permissions of 600 or 700 are suggested.

The **ustar** *typeflag* field was used to accommodate the additional functionality of the new format rather than magic or version because the POSIX.1-1990 standard (and, by reference, the previous version of *pax*), mandated the behavior of the format-reading utility when it encountered an unknown *typeflag*, but was silent about the other two fields.

Early proposals of the first revision to IEEE Std 1003.1-2001 contained a proposed archive format that was based on compatibility with the standard for tape files (ISO 1001, similar to the format used historically on many mainframes and minicomputers). This format was overly complex and required considerable overhead in volume and header records. Furthermore, the standard developers felt that it would not be acceptable to the community of POSIX developers, so it was later changed to be a format more closely related to historical practice on POSIX systems.

The prefix and name split of pathnames in **ustar** was replaced by the single path extended header record for simplicity.

The concept of a global extended header (*typeflag* **g**) was controversial. If this were applied to an archive being recorded on magnetic tape, a few unreadable blocks at the beginning of the tape could be a serious problem; a utility attempting to extract as many files as possible from a damaged archive could lose a large percentage of file header information in this case. However, if the archive were on a reliable medium, such as a CD-ROM, the global extended header offers considerable potential size reductions by eliminating redundant information. Thus, the text warns against using the global method for unreliable media and provides a method for implanting global information in the extended header for each file, rather than in the *typeflag* **g** records.

No facility for data translation or filtering on a per-file basis is included because the standard developers could not invent an interface that would allow this in an efficient manner. If a filter, such as encryption or compression, is to be applied to all the files, it is more efficient to apply the filter to the entire archive as a single file. The standard developers considered interfaces that would invoke a shell script for each file going into or out of the archive, but the system overhead in this approach was considered to be too high.

One such approach would be to have **filter=** records that give a pathname for an executable. When the program is invoked, the file and archive would be open for standard input/output and all the header fields would be available as environment variables or command-line arguments. The standard developers did discuss such schemes, but they were omitted from IEEE Std 1003.1-2001 due to

concerns about excessive overhead. Also, the program itself would need to be in the archive if it were to be used portably.

There is currently no portable means of identifying the character set(s) used for a file in the file system. Therefore, *pax* has not been given a mechanism to generate charset records automatically. The only portable means of doing this is for the user to write the archive using the **-o charset=** *string* command line option. This assumes that all of the files in the archive use the same encoding. The "implementation-defined" text is included to allow for a system that can identify the encodings used for each of its files.

The table of standards that accompanies the charset record description is acknowledged to be very limited. Only a limited number of character set standards is reasonable for maximal interchange. Any character set is, of course, possible by prior agreement. It was suggested that EBCDIC be listed, but it was omitted because it is not defined by a formal standard. Formal standards, and then only those with reasonably large followings, can be included here, simply as a matter of practicality. The <*value*>s represent names of officially registered character sets in the format required by the ISO 2375:1985 standard.

The normal comma or <blank>-separated list rules are not followed in the case of keyword options to allow ease of argument parsing for *getopts*.

Further information on character encodings is in pax Archive Character Set Encoding/Decoding .

The standard developers have reserved keyword name space for vendor extensions. It is suggested that the format to be used is:

VENDOR.keyword

where *VENDOR* is the name of the vendor or organization in all uppercase letters. It is further suggested that the keyword following the period be named differently than any of the standard keywords so that it could be used for future standardization, if appropriate, by omitting the *VENDOR* prefix.

The <*length*> field in the extended header record was included to make it simpler to step through the records, even if a record contains an unknown format (to a particular *pax*) with complex interactions of special characters. It also provides a minor integrity checkpoint within the records to aid a program attempting to recover files from a damaged archive.

There are no extended header versions of the *devmajor* and *devminor* fields because the unspecified format **ustar** header field should be sufficient. If they are not, vendor-specific extended keywords (such as *VENDOR.devmajor*) should be used.

Device and *i*-number labeling of files was not adopted from *cpio*; files are interchanged strictly on a symbolic name basis, as in **ustar**.

Just as with the **ustar** format descriptions, the new format makes no special arrangements for multi-volume archives. Each of the *pax* archive types is assumed to be inside a single POSIX file and splitting that file over multiple volumes (diskettes, tape cartridges, and so on), processing their labels, and mounting each in the proper sequence are considered to be implementation details that cannot be described portably.

The **pax** format is intended for interchange, not only for backup on a single (family of) systems. It is not as densely packed as might be possible for backup:

* It contains information as coded characters that could be coded in binary.
* It identifies extended records with name fields that could be omitted in favor of a fixed-field layout.
* It translates names into a portable character set and identifies locale-related information, both of which are probably unnecessary for backup.

The requirements on restoring from an archive are slightly different from the historical wording, allowing for non-monolithic privilege to bring forward as much as possible. In particular, attributes such as "high performance file" might be broadly but not universally granted while set-user-ID or *chown*() might be much more restricted. There is no implication in IEEE Std 1003.1-2001 that the security information be honored after it is restored to the file hierarchy, in spite of what might be improperly inferred by the silence on that topic. That is a topic for another standard.

Links are recorded in the fashion described here because a link can be to any file type. It is desirable in general to be able to restore part of an archive selectively and restore all of those files completely. If the data is not associated with each link, it is not possible to do this. However, the data associated with a file can be large, and when selective restoration is not needed, this can be a significant burden. The archive is structured so that files that have no associated data can always be restored by the name of any link name of any link, and the user may choose whether data is recorded with each instance of a file that contains data. The format permits mixing of both types of links in a single archive; this can be done for special needs, and *pax* is expected to interpret such archives on input properly, despite the fact that there is no *pax* option that would force this mixed case on output. (When **-o linkdata** is used, the output must contain the duplicate data, but the implementation is free to include it or omit it when **-o linkdata** is not used.)

The time values are included as extended header records for those implementations needing more than the eleven octal digits allowed by the **ustar** format. Portable file timestamps cannot be negative. If *pax* encounters a file with a negative timestamp in **copy** or **write** mode, it can reject the file, substitute a non-negative timestamp, or generate a non-portable timestamp with a leading **'-'** . Even though some implementations can support finer file-time granularities than seconds, the normative text requires support only for seconds since the Epoch because the ISO POSIX-1 standard states them that way. The **ustar** format includes only *mtime*; the new format adds *atime* and *ctime* for symmetry. The *atime* access time restored to the file system will be affected by the **-p a** and **-p e** options. The *ctime* creation time (actually *inode* modification time) is described with "appropriate privilege" so that it can be ignored when writing to the file system. POSIX does not provide a portable means to change file creation time. Nothing is intended to prevent a non-portable implementation of *pax* from restoring the value.

The *gid*, *size*, and *uid* extended header records were included to allow expansion beyond the sizes specified in the regular *tar* header. New file system architectures are emerging that will exhaust the 12-digit size field. There are probably not many systems requiring more than 8 digits for user and group IDs, but the extended header values were included for completeness, allowing overrides for all of the decimal values in the *tar* header.

The standard developers intended to describe the effective results of *pax* with regard to file ownerships and permissions; implementations are not restricted in timing or sequencing the restoration of such, provided the results are as specified.

Much of the text describing the extended headers refers to use in " **write** or **copy** modes". The **copy** mode references are due to the normative text: "The effect of the copy shall be as if the copied files were written to an archive file and then subsequently extracted ...". There is certainly no way to test whether *pax* is actually generating the extended headers in **copy** mode, but the effects must be as if it had.

pax Archive Character Set Encoding/Decoding

There is a need to exchange archives of files between systems of different native codesets. Filenames, group names, and user names must be preserved to the fullest extent possible when an archive is read on the receiving platform. Translation of the contents of files is not within the scope of the *pax* utility.

There will also be the need to represent characters that are not available on the receiving platform. These unsupported characters cannot be automatically folded to the local set of characters due to the chance of collisions. This could result in overwriting previous extracted files from the archive or pre-existing files on the system.

For these reasons, the codeset used to represent characters within the extended header records of the *pax* archive must be sufficiently rich to handle all commonly used character sets. The fields requiring translation include, at a minimum, filenames, user names, group names, and link pathnames. Implementations may wish to have localized extended keywords that use non-portable characters.

The standard developers considered the following options:

* The archive creator specifies the well-defined name of the source codeset. The receiver must then recognize the codeset name and perform the appropriate translations to the destination codeset.

* The archive creator includes within the archive the character mapping table for the source codeset used to encode extended header records. The receiver must then read the character mapping table and perform the appropriate translations to the destination codeset.

* The archive creator translates the extended header records in the source codeset into a canonical form. The receiver must then perform the appropriate translations to the destination codeset.

The approach that incorporates the name of the source codeset poses the problem of codeset name registration, and makes the archive useless to *pax* archive decoders that do not recognize that codeset.

Because parts of an archive may be corrupted, the standard developers felt that including the character map of the source codeset was too fragile. The loss of this one key component could result in making the entire archive useless. (The difference between this and the global extended header decision was that the latter has a workaround-duplicating extended header records on unreliable media-but this would be too burdensome for large character set maps.)

Both of the above approaches also put an undue burden on the *pax* archive receiver to handle the cross-product of all source and destination codesets.

To simplify the translation from the source codeset to the canonical form and from the canonical form to the destination codeset, the standard developers decided that the internal representation should be a stateless encoding. A stateless encoding is one where each codepoint has the same meaning, without regard to the decoder being in a specific state. An example of a stateful encoding would be the Japanese Shift-JIS; an example of a stateless encoding would be the ISO/IEC 646:1991 standard (equivalent to 7-bit ASCII).

For these reasons, the standard developers decided to adopt a canonical format for the representation of file information strings. The obvious, well-endorsed candidate is the ISO/IEC 10646-1:2000 standard (based in part on Unicode), which can be used to represent the characters of virtually all standardized character sets. The standard developers initially agreed upon using UCS2 (16-bit Unicode) as the internal representation. This repertoire of characters provides a sufficiently rich set to represent all commonly-used codesets.

However, the standard developers found that the 16-bit Unicode representation had some problems. It forced the issue of standardizing byte ordering. The 2-byte length of each character made the extended header records twice as long for the case of strings coded entirely from historical 7-bit ASCII. For these reasons, the standard developers chose the UTF-8 defined in the ISO/IEC 10646-1:2000 standard. This multi-byte representation encodes UCS2 or UCS4 characters reliably and deterministically, eliminating the need for a canonical byte ordering. In addition, NUL octets and other characters possibly confusing to POSIX file systems do not appear, except to represent themselves. It was realized that certain national codesets take up more space after the encoding, due to their placement within the UCS range; it was felt that the usefulness of the encoding of the names outweighs the disadvantage of size increase for file, user, and group names.

The encoding of UTF-8 is as follows:

UCS4 Hex Encoding UTF-8 Binary Encoding

00000000-0000007F 0xxxxxxx
00000080-000007FF 110xxxxx 10xxxxxx
00000800-0000FFFF 1110xxxx 10xxxxxx 10xxxxxx
00010000-001FFFFF 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000-03FFFFFF 111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000-7FFFFFFF 1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx

where each **'x'** represents a bit value from the character being translated.

ustar Interchange Format

The description of the **ustar** format reflects numerous enhancements over pre-1988 versions of the historical *tar* utility. The goal of these changes was not only to provide the functional enhancements desired, but also to retain compatibility between new and old versions. This compatibility has been retained. Archives written using the old archive format are compatible with the new format.

Implementors should be aware that the previous file format did not include a mechanism to archive directory type files. For this reason, the convention of using a filename ending with slash was adopted to specify a directory on the archive.

The total size of the *name* and *prefix* fields have been set to meet the minimum requirements for {PATH_MAX}. If a pathname will fit within the *name* field, it is recommended that the pathname be

stored there without the use of the *prefix* field. Although the name field is known to be too small to contain {PATH_MAX} characters, the value was not changed in this version of the archive file format to retain backwards-compatibility, and instead the prefix was introduced. Also, because of the earlier version of the format, there is no way to remove the restriction on the *linkname* field being limited in size to just that of the *name* field.

The *size* field is required to be meaningful in all implementation extensions, although it could be zero. This is required so that the data blocks can always be properly counted.

It is suggested that if device special files need to be represented that cannot be represented in the standard format, that one of the extension types (**A**- **Z**) be used, and that the additional information for the special file be represented as data and be reflected in the *size* field.

Attempting to restore a special file type, where it is converted to ordinary data and conflicts with an existing filename, need not be specially detected by the utility. If run as an ordinary user, *pax* should not be able to overwrite the entries in, for example, **/dev** in any case (whether the file is converted to another type or not). If run as a privileged user, it should be able to do so, and it would be considered a bug if it did not. The same is true of ordinary data files and similarly named special files; it is impossible to anticipate the needs of the user (who could really intend to overwrite the file), so the behavior should be predictable (and thus regular) and rely on the protection system as required.

The value 7 in the *typeflag* field is intended to define how contiguous files can be stored in a **ustar** archive. IEEE Std 1003.1-2001 does not require the contiguous file extension, but does define a standard way of archiving such files so that all conforming systems can interpret these file types in a meaningful and consistent manner. On a system that does not support extended file types, the *pax* utility should do the best it can with the file and go on to the next.

The file protection modes are those conventionally used by the *ls* utility. This is extended beyond the usage in the ISO POSIX-2 standard to support the "shared text" or "sticky" bit. It is intended that the conformance document should not document anything beyond the existence of and support of such a mode. Further extensions are expected to these bits, particularly with overloading the set-user-ID and set-group-ID flags.

cpio Interchange Format

The reference to appropriate privilege in the **cpio** format refers to an error on standard output; the **ustar** format does not make comparable statements.

The model for this format was the historical System V *cpio* **-c** data interchange format. This model documents the portable version of the **cpio** format and not the binary version. It has the flexibility to transfer data of any type described within IEEE Std 1003.1-2001, yet is extensible to transfer data types specific to extensions beyond IEEE Std 1003.1-2001 (for example, contiguous files). Because it describes existing practice, there is no question of maintaining upwards-compatibility.

cpio Header

There has been some concern that the size of the *c_ino* field of the header is too small to handle those systems that have very large *inode* numbers. However, the *c_ino* field in the header is used strictly as a hard-link resolution mechanism for archives. It is not necessarily the same value as the *inode* number of the file in the location from which that file is extracted.

The name *c_magic* is based on historical usage.

cpio Filename

For most historical implementations of the *cpio* utility, {PATH_MAX} octets can be used to describe the pathname without the addition of any other header fields (the NUL character would be included in this count). {PATH_MAX} is the minimum value for pathname size, documented as 256 bytes. However, an implementation may use *c_namesize* to determine the exact length of the pathname. With the current description of the <*cpio.h*> header, this pathname size can be as large as a number that is described in six octal digits.

Two values are documented under the *c_mode* field values to provide for extensibility for known file types:

0110 000

Reserved for contiguous files. The implementation may treat the rest of the information for

this archive like a regular file. If this file type is undefined, the implementation may create the file as a regular file.

This provides for extensibility of the **cpio** format while allowing for the ability to read old archives. Files of an unknown type may be read as "regular files" on some implementations. On a system that does not support extended file types, the *pax* utility should do the best it can with the file and go on to the next.

FUTURE DIRECTIONS

None.

SEE ALSO

Shell Command Language, *cp*, *ed*, *getopts*, *ls*, *printf*(), the Base Definitions volume of IEEE Std 1003.1-2001, <*cpio.h*>, the System Interfaces volume of IEEE Std 1003.1-2001, *chown*(), *creat*(), *mkdir*(), *mkfifo*(), *stat*(), *utime*(), *write*()

COPYRIGHT

NAME

archive_entry_acl_add_entry, **archive_entry_acl_add_entry_w**, **archive_entry_acl_clear**, **archive_entry_acl_count**, **archive_entry_acl_next**, **archive_entry_acl_next_w**, **archive_entry_acl_reset**, **archive_entry_acl_text_w**, **archive_entry_atime**, **archive_entry_atime_nsec**, **archive_entry_clear**, **archive_entry_clone**, **archive_entry_copy_fflags_text_w**, **archive_entry_copy_gname**, **archive_entry_copy_gname_w**, **archive_entry_copy_hardlink**, **archive_entry_copy_hardlink_w**, **archive_entry_copy_pathname_w**, **archive_entry_copy_stat**, **archive_entry_copy_symlink**, **archive_entry_copy_symlink_w**, **archive_entry_copy_uname**, **archive_entry_copy_uname_w**, **archive_entry_dev**, **archive_entry_devmajor**, **archive_entry_devminor**, **archive_entry_filetype**, **archive_entry_fflags**, **archive_entry_fflags_text**, **archive_entry_free**, **archive_entry_gid**, **archive_entry_gname**, **archive_entry_hardlink**, **archive_entry_ino**, **archive_entry_mode**, **archive_entry_mtime**, **archive_entry_mtime_nsec**, **archive_entry_nlink**, **archive_entry_new**, **archive_entry_pathname**, **archive_entry_pathname_w**, **archive_entry_rdev**, **archive_entry_rdevmajor**, **archive_entry_rdevminor**, **archive_entry_set_atime**, **archive_entry_set_ctime**, **archive_entry_set_dev**, **archive_entry_set_devmajor**, **archive_entry_set_devminor**, **archive_entry_set_filetype**, **archive_entry_set_fflags**, **archive_entry_set_gid**, **archive_entry_set_gname**, **archive_entry_set_hardlink**, **archive_entry_set_link**, **archive_entry_set_mode**, **archive_entry_set_mtime**, **archive_entry_set_pathname**, **archive_entry_set_rdevmajor**, **archive_entry_set_rdevminor**, **archive_entry_set_size**, **archive_entry_set_symlink**, **archive_entry_set_uid**, **archive_entry_set_uname**, **archive_entry_size**, **archive_entry_stat**, **archive_entry_symlink**, **archive_entry_uid**, **archive_entry_uname** — functions for manipulating archive entry descriptions

SYNOPSIS

```
#include <archive_entry.h>

void
archive_entry_acl_add_entry(struct archive_entry *, int type,
    int permset, int tag, int qual, const char *name);

void
archive_entry_acl_add_entry_w(struct archive_entry *, int type,
    int permset, int tag, int qual, const wchar_t *name);

void
archive_entry_acl_clear(struct archive_entry *);

int
archive_entry_acl_count(struct archive_entry *, int type);

int
archive_entry_acl_next(struct archive_entry *, int want_type, int *type,
    int *permset, int *tag, int *qual, const char **name);

int
archive_entry_acl_next_w(struct archive_entry *, int want_type,
    int *type, int *permset, int *tag, int *qual, const wchar_t **name);

int
archive_entry_acl_reset(struct archive_entry *, int want_type);
```

```
const wchar_t *
archive_entry_acl_text_w(struct archive_entry *, int flags);

time_t
archive_entry_atime(struct archive_entry *);

long
archive_entry_atime_nsec(struct archive_entry *);

struct archive_entry *
archive_entry_clear(struct archive_entry *);

struct archive_entry *
archive_entry_clone(struct archive_entry *);

const wchar_t *
archive_entry_copy_fflags_text_w(struct archive_entry *,
    const wchar_t *);

void
archive_entry_copy_gname(struct archive_entry *, const char *);

void
archive_entry_copy_gname_w(struct archive_entry *, const wchar_t *);

void
archive_entry_copy_hardlink(struct archive_entry *, const char *);

void
archive_entry_copy_hardlink_w(struct archive_entry *, const wchar_t *);

void
archive_entry_copy_pathname_w(struct archive_entry *, const wchar_t *);

void
archive_entry_copy_stat(struct archive_entry *, const struct stat *);

void
archive_entry_copy_symlink(struct archive_entry *, const char *);

void
archive_entry_copy_symlink_w(struct archive_entry *, const wchar_t *);

void
archive_entry_copy_uname(struct archive_entry *, const char *);

void
archive_entry_copy_uname_w(struct archive_entry *, const wchar_t *);

dev_t
archive_entry_dev(struct archive_entry *);

dev_t
archive_entry_devmajor(struct archive_entry *);

dev_t
archive_entry_devminor(struct archive_entry *);

mode_t
archive_entry_filetype(struct archive_entry *);

void
archive_entry_fflags(struct archive_entry *, unsigned long *set,
    unsigned long *clear);
```

*const char **
archive_entry_fflags_text(*struct archive_entry **);

void
archive_entry_free(*struct archive_entry **);

*const char **
archive_entry_gname(*struct archive_entry **);

*const char **
archive_entry_hardlink(*struct archive_entry **);

ino_t
archive_entry_ino(*struct archive_entry **);

mode_t
archive_entry_mode(*struct archive_entry **);

time_t
archive_entry_mtime(*struct archive_entry **);

long
archive_entry_mtime_nsec(*struct archive_entry **);

unsigned int
archive_entry_nlink(*struct archive_entry **);

*struct archive_entry **
archive_entry_new(*void*);

*const char **
archive_entry_pathname(*struct archive_entry **);

*const wchar_t **
archive_entry_pathname_w(*struct archive_entry **);

dev_t
archive_entry_rdev(*struct archive_entry **);

dev_t
archive_entry_rdevmajor(*struct archive_entry **);

dev_t
archive_entry_rdevminor(*struct archive_entry **);

void
archive_entry_set_dev(*struct archive_entry **, *dev_t*);

void
archive_entry_set_devmajor(*struct archive_entry **, *dev_t*);

void
archive_entry_set_devminor(*struct archive_entry **, *dev_t*);

void
archive_entry_set_filetype(*struct archive_entry **, *unsigned int*);

void
archive_entry_set_fflags(*struct archive_entry **, *unsigned long set*,
unsigned long clear);

void
archive_entry_set_gid(*struct archive_entry **, *gid_t*);

void
archive_entry_set_gname(*struct archive_entry **, *const char **);

void
archive_entry_set_hardlink(*struct archive_entry* *, *const char* *);

void
archive_entry_set_ino(*struct archive_entry* *, *unsigned long*);

void
archive_entry_set_link(*struct archive_entry* *, *const char* *);

void
archive_entry_set_mode(*struct archive_entry* *, *mode_t*);

void
archive_entry_set_mtime(*struct archive_entry* *, *time_t*, *long nanos*);

void
archive_entry_set_nlink(*struct archive_entry* *, *unsigned int*);

void
archive_entry_set_pathname(*struct archive_entry* *, *const char* *);

void
archive_entry_set_rdev(*struct archive_entry* *, *dev_t*);

void
archive_entry_set_rdevmajor(*struct archive_entry* *, *dev_t*);

void
archive_entry_set_rdevminor(*struct archive_entry* *, *dev_t*);

void
archive_entry_set_size(*struct archive_entry* *, *int64_t*);

void
archive_entry_set_symlink(*struct archive_entry* *, *const char* *);

void
archive_entry_set_uid(*struct archive_entry* *, *uid_t*);

void
archive_entry_set_uname(*struct archive_entry* *, *const char* *);

int64_t
archive_entry_size(*struct archive_entry* *);

const struct stat *
archive_entry_stat(*struct archive_entry* *);

const char *
archive_entry_symlink(*struct archive_entry* *);

const char *
archive_entry_uname(*struct archive_entry* *);

DESCRIPTION

These functions create and manipulate data objects that represent entries within an archive. You can think of a struct archive_entry as a heavy-duty version of struct stat: it includes everything from struct stat plus associated pathname, textual group and user names, etc. These objects are used by libarchive(3) to represent the metadata associated with a particular entry in an archive.

Create and Destroy

There are functions to allocate, destroy, clear, and copy *archive_entry* objects:

archive_entry_clear()
: Erases the object, resetting all internal fields to the same state as a newly-created object. This is provided to allow you to quickly recycle objects without thrashing the heap.

archive_entry_clone()
: A deep copy operation; all text fields are duplicated.

archive_entry_free()
: Releases the struct archive_entry object.

archive_entry_new()
: Allocate and return a blank struct archive_entry object.

Set and Get Functions

Most of the functions here set or read entries in an object. Such functions have one of the following forms:

archive_entry_set_XXXX()
: Stores the provided data in the object. In particular, for strings, the pointer is stored, not the referenced string.

archive_entry_copy_XXXX()
: As above, except that the referenced data is copied into the object.

archive_entry_XXXX()
: Returns the specified data. In the case of strings, a const-qualified pointer to the string is returned.

String data can be set or accessed as wide character strings or normal *char* strings. The functions that use wide character strings are suffixed with **_w**. Note that these are different representations of the same data: For example, if you store a narrow string and read the corresponding wide string, the object will transparently convert formats using the current locale. Similarly, if you store a wide string and then store a narrow string for the same data, the previously-set wide string will be discarded in favor of the new data.

There are a few set/get functions that merit additional description:

archive_entry_set_link()
: This function sets the symlink field if it is already set. Otherwise, it sets the hardlink field.

File Flags

File flags are transparently converted between a bitmap representation and a textual format. For example, if you set the bitmap and ask for text, the library will build a canonical text format. However, if you set a text format and request a text format, you will get back the same text, even if it is ill-formed. If you need to canonicalize a textual flags string, you should first set the text form, then request the bitmap form, then use that to set the bitmap form. Setting the bitmap format will clear the internal text representation and force it to be reconstructed when you next request the text form.

The bitmap format consists of two integers, one containing bits that should be set, the other specifying bits that should be cleared. Bits not mentioned in either bitmap will be ignored. Usually, the bitmap of bits to be cleared will be set to zero. In unusual circumstances, you can force a fully-specified set of file flags by setting the bitmap of flags to clear to the complement of the bitmap of flags to set. (This differs from `fflagstostr`(3), which only includes names for set bits.) Converting a bitmap to a textual string is a platform-specific operation; bits that are not meaningful on the current platform will be ignored.

The canonical text format is a comma-separated list of flag names. The **archive_entry_copy_fflags_text_w**() function parses the provided text and sets the internal bitmap values. This is a platform-specific operation; names that are not meaningful on the current platform will be ignored. The function returns a pointer to the start of the first name that was not recognized, or NULL if every name was recognized. Note that every name--including names that follow an unrecognized name--will be evaluated, and the bitmaps will be set to reflect every name that is recognized. (In particular, this differs from `strtofflags`(3), which stops parsing at the first unrecognized name.)

ACL Handling

XXX This needs serious help. XXX

An "Access Control List" (ACL) is a list of permissions that grant access to particular users or groups beyond what would normally be provided by standard POSIX mode bits. The ACL handling here addresses some deficiencies in the POSIX.1e draft 17 ACL specification. In particular, POSIX.1e draft 17 specifies several different formats, but none of those formats include both textual user/group names and numeric UIDs/GIDs.

XXX explain ACL stuff XXX

SEE ALSO

`archive`(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩. archive_read

NAME

archive_read_new, **archive_read_support_compression_all**, **archive_read_support_compression_bzip2**, **archive_read_support_compression_compress**, **archive_read_support_compression_gzip**, **archive_read_support_compression_none**, **archive_read_support_compression_program**, **archive_read_support_format_all**, **archive_read_support_format_cpio**, **archive_read_support_format_empty**, **archive_read_support_format_iso9660**, **archive_read_support_format_tar**, **archive_read_support_format_zip**, **archive_read_open**, **archive_read_open2**, **archive_read_open_fd**, **archive_read_open_FILE**, **archive_read_open_filename**, **archive_read_open_memory**, **archive_read_next_header**, **archive_read_data**, **archive_read_data_block**, **archive_read_data_skip**, **archive_read_data_into_buffer**, **archive_read_data_into_fd**, **archive_read_extract**, **archive_read_extract_set_progress_callback**, **archive_read_close**, **archive_read_finish** — functions for reading streaming archives

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_read_new(void);

int
archive_read_support_compression_all(struct archive *);

int
archive_read_support_compression_bzip2(struct archive *);

int
archive_read_support_compression_compress(struct archive *);

int
archive_read_support_compression_gzip(struct archive *);

int
archive_read_support_compression_none(struct archive *);

int
archive_read_support_compression_program(struct archive *,
    const char *cmd);

int
archive_read_support_format_all(struct archive *);

int
archive_read_support_format_cpio(struct archive *);

int
archive_read_support_format_empty(struct archive *);

int
archive_read_support_format_iso9660(struct archive *);

int
archive_read_support_format_tar(struct archive *);

int
archive_read_support_format_zip(struct archive *);
```

```
int
archive_read_open(struct archive *, void *client_data,
    archive_open_callback *, archive_read_callback *,
    archive_close_callback *);

int
archive_read_open2(struct archive *, void *client_data,
    archive_open_callback *, archive_read_callback *,
    archive_skip_callback *, archive_close_callback *);

int
archive_read_open_FILE(struct archive *, FILE *file);

int
archive_read_open_fd(struct archive *, int fd, size_t block_size);

int
archive_read_open_filename(struct archive *, const char *filename,
    size_t block_size);

int
archive_read_open_memory(struct archive *, void *buff, size_t size);

int
archive_read_next_header(struct archive *, struct archive_entry **);

ssize_t
archive_read_data(struct archive *, void *buff, size_t len);

int
archive_read_data_block(struct archive *, const void **buff,
    size_t *len, off_t *offset);

int
archive_read_data_skip(struct archive *);

int
archive_read_data_into_buffer(struct archive *, void *, ssize_t len);

int
archive_read_data_into_fd(struct archive *, int fd);

int
archive_read_extract(struct archive *, struct archive_entry *,
    int flags);

void
archive_read_extract_set_progress_callback(struct archive *,
    void (*func)(void *), void *user_data);

int
archive_read_close(struct archive *);

int
archive_read_finish(struct archive *);
```

DESCRIPTION

These functions provide a complete API for reading streaming archives. The general process is to first create the struct archive object, set options, initialize the reader, iterate over the archive headers and associated data, then close the archive and release all resources. The following summary describes the functions in approximately the order they would be used:

archive_read_new()
: Allocates and initializes a struct archive object suitable for reading from an archive.

archive_read_support_compression_all(),
archive_read_support_compression_bzip2(),
archive_read_support_compression_compress(),
archive_read_support_compression_gzip(),
archive_read_support_compression_none()
: Enables auto-detection code and decompression support for the specified compression. Note that "none" is always enabled by default. For convenience, **archive_read_support_compression_all**() enables all available decompression code.

archive_read_support_compression_program()
: Data is fed through the specified external program before being dearchived. Note that this disables automatic detection of the compression format, so it makes no sense to specify this in conjunction with any other decompression option.

archive_read_support_format_all(), **archive_read_support_format_cpio**(),
archive_read_support_format_empty(),
archive_read_support_format_iso9660(),
archive_read_support_format_tar,()
archive_read_support_format_zip()
: Enables support---including auto-detection code---for the specified archive format. For example, **archive_read_support_format_tar**() enables support for a variety of standard tar formats, old-style tar, ustar, pax interchange format, and many common variants. For convenience, **archive_read_support_format_all**() enables support for all available formats. Only empty archives are supported by default.

archive_read_open()
: The same as **archive_read_open2**(), except that the skip callback is assumed to be NULL.

archive_read_open2()
: Freeze the settings, open the archive, and prepare for reading entries. This is the most generic version of this call, which accepts four callback functions. Most clients will want to use **archive_read_open_filename**(), **archive_read_open_FILE**(), **archive_read_open_fd**(), or **archive_read_open_memory**() instead. The library invokes the client-provided functions to obtain raw bytes from the archive.

archive_read_open_FILE()
: Like **archive_read_open**(), except that it accepts a *FILE* * pointer. This function should not be used with tape drives or other devices that require strict I/O blocking.

archive_read_open_fd()
: Like **archive_read_open**(), except that it accepts a file descriptor and block size rather than a set of function pointers. Note that the file descriptor will not be automatically closed at end-of-archive. This function is safe for use with tape drives or other blocked devices.

archive_read_open_file()
: This is a deprecated synonym for **archive_read_open_filename**().

archive_read_open_filename()
: Like **archive_read_open**(), except that it accepts a simple filename and a block size. A NULL filename represents standard input. This function is safe for use with tape drives or other blocked devices.

archive_read_open_memory()
: Like **archive_read_open**(), except that it accepts a pointer and size of a block of memory containing the archive data.

archive_read_next_header()
: Read the header for the next entry and return a pointer to a struct archive_entry.

archive_read_data()
: Read data associated with the header just read. Internally, this is a convenience function that calls **archive_read_data_block**() and fills any gaps with nulls so that callers see a single continuous stream of data.

archive_read_data_block()
: Return the next available block of data for this entry. Unlike **archive_read_data**(), the **archive_read_data_block**() function avoids copying data and allows you to correctly handle sparse files, as supported by some archive formats. The library guarantees that offsets will increase and that blocks will not overlap. Note that the blocks returned from this function can be much larger than the block size read from disk, due to compression and internal buffer optimizations.

archive_read_data_skip()
: A convenience function that repeatedly calls **archive_read_data_block**() to skip all of the data for this archive entry.

archive_read_data_into_buffer()
: This function is deprecated and will be removed. Use **archive_read_data**() instead.

archive_read_data_into_fd()
: A convenience function that repeatedly calls **archive_read_data_block**() to copy the entire entry to the provided file descriptor.

archive_read_extract(), **archive_read_extract_set_skip_file**()
: A convenience function that wraps the corresponding `archive_write_disk`(3) interfaces. The first call to **archive_read_extract**() creates a restore object using `archive_write_disk_new`(3) and `archive_write_disk_set_standard_lookup`(3), then transparently invokes `archive_write_disk_set_options`(3), `archive_write_header`(3), `archive_write_data`(3), and `archive_write_finish_entry`(3) to create the entry on disk and copy data into it. The *flags* argument is passed unmodified to `archive_write_disk_set_options`(3).

archive_read_extract_set_progress_callback()
: Sets a pointer to a user-defined callback that can be used for updating progress displays during extraction. The progress function will be invoked during the extraction of large regular files. The progress function will be invoked with the pointer provided to this call. Generally, the data pointed to should include a reference to the archive object and the archive_entry object so that various statistics can be retrieved for the progress display.

archive_read_close()
: Complete the archive and invoke the close callback.

archive_read_finish()
: Invokes **archive_read_close**() if it was not invoked manually, then release all resources. Note: In libarchive 1.x, this function was declared to return *void*, which made it impossible to detect certain errors when **archive_read_close**() was invoked implicitly from this function. The declaration is corrected beginning with libarchive 2.0.

Note that the library determines most of the relevant information about the archive by inspection. In particular, it automatically detects `gzip`(1) or `bzip2`(1) compression and transparently performs the appropriate decompression. It also automatically detects the archive format.

A complete description of the struct archive and struct archive_entry objects can be found in the overview manual page for `libarchive`(3).

CLIENT CALLBACKS

The callback functions must match the following prototypes:

typedef ssize_t **archive_read_callback**(*struct archive *, void *client_data, const void **buffer*)

typedef int **archive_skip_callback**(*struct archive *, void *client_data, size_t request*)

typedef int **archive_open_callback**(*struct archive *, void *client_data*)

typedef int **archive_close_callback**(*struct archive *, void *client_data*)

The open callback is invoked by **archive_open**(). It should return **ARCHIVE_OK** if the underlying file or data source is successfully opened. If the open fails, it should call **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL**.

The read callback is invoked whenever the library requires raw bytes from the archive. The read callback should read data into a buffer, set the `const void **buffer` argument to point to the available data, and return a count of the number of bytes available. The library will invoke the read callback again only after it has consumed this data. The library imposes no constraints on the size of the data blocks returned. On end-of-file, the read callback should return zero. On error, the read callback should invoke **archive_set_error**() to register an error code and message and return -1.

The skip callback is invoked when the library wants to ignore a block of data. The return value is the number of bytes actually skipped, which may differ from the request. If the callback cannot skip data, it should return zero. If the skip callback is not provided (the function pointer is `NULL`), the library will invoke the read function instead and simply discard the result. A skip callback can provide significant performance gains when reading uncompressed archives from slow disk drives or other media that can skip quickly.

The close callback is invoked by archive_close when the archive processing is complete. The callback should return **ARCHIVE_OK** on success. On failure, the callback should invoke **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL.**

EXAMPLE

The following illustrates basic usage of the library. In this example, the callback functions are simply wrappers around the standard `open`(2), `read`(2), and `close`(2) system calls.

```
void
list_archive(const char *name)
{
  struct mydata *mydata;
  struct archive *a;
  struct archive_entry *entry;

  mydata = malloc(sizeof(struct mydata));
  a = archive_read_new();
  mydata->name = name;
  archive_read_support_compression_all(a);
  archive_read_support_format_all(a);
  archive_read_open(a, mydata, myopen, myread, myclose);
  while (archive_read_next_header(a, &entry) == ARCHIVE_OK) {
    printf("%s\n",archive_entry_pathname(entry));
    archive_read_data_skip(a);
  }
  archive_read_finish(a);
  free(mydata);
}

ssize_t
myread(struct archive *a, void *client_data, const void **buff)
{
  struct mydata *mydata = client_data;

  *buff = mydata->buff;
  return (read(mydata->fd, mydata->buff, 10240));
}
```

```
int
myopen(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  mydata->fd = open(mydata->name, O_RDONLY);
  return (mydata->fd >= 0 ? ARCHIVE_OK : ARCHIVE_FATAL);
}

int
myclose(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  if (mydata->fd > 0)
    close(mydata->fd);
  return (ARCHIVE_OK);
}
```

RETURN VALUES

Most functions return zero on success, non-zero on error. The possible return codes include: **ARCHIVE_OK** (the operation succeeded), **ARCHIVE_WARN** (the operation succeeded but a non-critical error was encountered), **ARCHIVE_EOF** (end-of-archive was encountered), **ARCHIVE_RETRY** (the operation failed but can be retried), and **ARCHIVE_FATAL** (there was a fatal error; the archive should be closed immediately). Detailed error codes and textual descriptions are available from the **archive_errno**() and **archive_error_string**() functions.

archive_read_new() returns a pointer to a freshly allocated struct archive object. It returns NULL on error.

archive_read_data() returns a count of bytes actually read or zero at the end of the entry. On error, a value of **ARCHIVE_FATAL**, **ARCHIVE_WARN**, or **ARCHIVE_RETRY** is returned and an error code and textual description can be retrieved from the **archive_errno**() and **archive_error_string**() functions.

The library expects the client callbacks to behave similarly. If there is an error, you can use **archive_set_error**() to set an appropriate error code and description, then return one of the non-zero values above. (Note that the value eventually returned to the client may not be the same; many errors that are not critical at the level of basic I/O can prevent the archive from being properly read, thus most I/O errors eventually cause **ARCHIVE_FATAL** to be returned.)

SEE ALSO

tar(1), archive(3), archive_util(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Many traditional archiver programs treat empty files as valid empty archives. For example, many implementations of tar(1) allow you to append entries to an empty file. Of course, it is impossible to determine the format of an empty file by inspecting the contents, so this library treats empty files as having a special “empty” format. archive_util

NAME

archive_clear_error, **archive_compression**, **archive_compression_name**, **archive_copy_error**, **archive_errno**, **archive_error_string**, **archive_format**, **archive_format_name**, **archive_set_error** — libarchive utility functions

SYNOPSIS

```
#include <archive.h>

void
archive_clear_error(struct archive *);

int
archive_compression(struct archive *);

const char *
archive_compression_name(struct archive *);

void
archive_copy_error(struct archive *, struct archive *);

int
archive_errno(struct archive *);

const char *
archive_error_string(struct archive *);

int
archive_format(struct archive *);

const char *
archive_format_name(struct archive *);

void
archive_set_error(struct archive *, int error_code, const char *fmt,
    ...);
```

DESCRIPTION

These functions provide access to various information about the struct archive object used in the libarchive(3) library.

archive_clear_error()
: Clears any error information left over from a previous call. Not generally used in client code.

archive_compression()
: Returns a numeric code indicating the current compression. This value is set by **archive_read_open**().

archive_compression_name()
: Returns a text description of the current compression suitable for display.

archive_copy_error()
: Copies error information from one archive to another.

archive_errno()
: Returns a numeric error code (see errno(2)) indicating the reason for the most recent error return.

archive_error_string()
: Returns a textual error message suitable for display. The error message here is usually more specific than that obtained from passing the result of **archive_errno**() to strerror(3).

archive_format()
: Returns a numeric code indicating the format of the current archive entry. This value is set by a successful call to **archive_read_next_header**(). Note that it is common for this value to change from entry to entry. For example, a tar archive might have several entries that utilize GNU tar extensions and several entries that do not. These entries will have different format codes.

archive_format_name()
A textual description of the format of the current entry.

archive_set_error()
Sets the numeric error code and error description that will be returned by **archive_errno**() and **archive_error_string**(). This function should be used within I/O callbacks to set system-specific error codes and error descriptions. This function accepts a printf-like format string and arguments. However, you should be careful to use only the following printf format specifiers: “%c”, “%d”, “%jd”, “%jo”, “%ju”, “%jx”, “%ld”, “%lo”, “%lu”, “%lx”, “%o”, “%u”, “%s”, “%x”, “%%”. Field-width specifiers and other printf features are not uniformly supported and should not be used.

SEE ALSO

archive_read(3), archive_write(3), libarchive(3), printf(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩. archive_write

NAME

archive_write_new, **archive_write_set_format_cpio**, **archive_write_set_format_pax**, **archive_write_set_format_pax_restricted**, **archive_write_set_format_shar**, **archive_write_set_format_shar_binary**, **archive_write_set_format_ustar**, **archive_write_get_bytes_per_block**, **archive_write_set_bytes_per_block**, **archive_write_set_bytes_in_last_block**, **archive_write_set_compression_bzip2**, **archive_write_set_compression_gzip**, **archive_write_set_compression_none**, **archive_write_set_compression_program**, **archive_write_open**, **archive_write_open_fd**, **archive_write_open_FILE**, **archive_write_open_filename**, **archive_write_open_memory**, **archive_write_header**, **archive_write_data**, **archive_write_finish_entry**, **archive_write_close**, **archive_write_finish** — functions for creating archives

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_write_new(void);

int
archive_write_get_bytes_per_block(struct archive *);

int
archive_write_set_bytes_per_block(struct archive *,
    int bytes_per_block);

int
archive_write_set_bytes_in_last_block(struct archive *, int);

int
archive_write_set_compression_bzip2(struct archive *);

int
archive_write_set_compression_gzip(struct archive *);

int
archive_write_set_compression_none(struct archive *);

int
archive_write_set_compression_program(struct archive *,
    const char * cmd);

int
archive_write_set_format_cpio(struct archive *);

int
archive_write_set_format_pax(struct archive *);

int
archive_write_set_format_pax_restricted(struct archive *);

int
archive_write_set_format_shar(struct archive *);

int
archive_write_set_format_shar_binary(struct archive *);

int
archive_write_set_format_ustar(struct archive *);
```

```
int
archive_write_open(struct archive *, void *client_data,
    archive_open_callback *, archive_write_callback *,
    archive_close_callback *);

int
archive_write_open_fd(struct archive *, int fd);

int
archive_write_open_FILE(struct archive *, FILE *file);

int
archive_write_open_filename(struct archive *, const char *filename);

int
archive_write_open_memory(struct archive *, void *buffer,
    size_t bufferSize, size_t *outUsed);

int
archive_write_header(struct archive *, struct archive_entry *);

ssize_t
archive_write_data(struct archive *, const void *, size_t);

int
archive_write_finish_entry(struct archive *);

int
archive_write_close(struct archive *);

int
archive_write_finish(struct archive *);
```

DESCRIPTION

These functions provide a complete API for creating streaming archive files. The general process is to first create the struct archive object, set any desired options, initialize the archive, append entries, then close the archive and release all resources. The following summary describes the functions in approximately the order they are ordinarily used:

archive_write_new()
: Allocates and initializes a struct archive object suitable for writing a tar archive.

archive_write_set_bytes_per_block()
: Sets the block size used for writing the archive data. Every call to the write callback function, except possibly the last one, will use this value for the length. The third parameter is a boolean that specifies whether or not the final block written will be padded to the full block size. If it is zero, the last block will not be padded. If it is non-zero, padding will be added both before and after compression. The default is to use a block size of 10240 bytes and to pad the last block. Note that a block size of zero will suppress internal blocking and cause writes to be sent directly to the write callback as they occur.

archive_write_get_bytes_per_block()
: Retrieve the block size to be used for writing. A value of -1 here indicates that the library should use default values. A value of zero indicates that internal blocking is suppressed.

archive_write_set_bytes_in_last_block()
: Sets the block size used for writing the last block. If this value is zero, the last block will be padded to the same size as the other blocks. Otherwise, the final block will be padded to a multiple of this size. In particular, setting it to 1 will cause the final block to not be padded. For compressed output, any padding generated by this option is applied only after the compression. The uncompressed data is always unpadded. The default is to pad the last block to the full block size (note that **archive_write_open_filename**() will set this based on the file type). Unlike the other "set" functions, this function can be called after the archive is

opened.

archive_write_get_bytes_in_last_block()
Retrieve the currently-set value for last block size. A value of -1 here indicates that the library should use default values.

archive_write_set_format_cpio(), **archive_write_set_format_pax**(), **archive_write_set_format_pax_restricted**(), **archive_write_set_format_shar**(), **archive_write_set_format_shar_binary**(), **archive_write_set_format_ustar**()
Sets the format that will be used for the archive. The library can write POSIX octet-oriented cpio format archives, POSIX-standard "pax interchange" format archives, traditional "shar" archives, enhanced "binary" shar archives that store a variety of file attributes and handle binary files, and POSIX-standard "ustar" archives. The pax interchange format is a backwards-compatible tar format that adds key/value attributes to each entry and supports arbitrary filenames, linknames, uids, sizes, etc. "Restricted pax interchange format" is the library default; this is the same as pax format, but suppresses the pax extended header for most normal files. In most cases, this will result in ordinary ustar archives.

archive_write_set_compression_bzip2(), **archive_write_set_compression_gzip**(), **archive_write_set_compression_none**()
The resulting archive will be compressed as specified. Note that the compressed output is always properly blocked.

archive_write_set_compression_program()
The archive will be fed into the specified compression program. The output of that program is blocked and written to the client write callbacks.

archive_write_open()
Freeze the settings, open the archive, and prepare for writing entries. This is the most generic form of this function, which accepts pointers to three callback functions which will be invoked by the compression layer to write the constructed archive.

archive_write_open_fd()
A convenience form of **archive_write_open**() that accepts a file descriptor. The **archive_write_open_fd**() function is safe for use with tape drives or other block-oriented devices.

archive_write_open_FILE()
A convenience form of **archive_write_open**() that accepts a *FILE* * pointer. Note that **archive_write_open_FILE**() is not safe for writing to tape drives or other devices that require correct blocking.

archive_write_open_file()
A deprecated synonym for **archive_write_open_filename**().

archive_write_open_filename()
A convenience form of **archive_write_open**() that accepts a filename. A NULL argument indicates that the output should be written to standard output; an argument of "-" will open a file with that name. If you have not invoked **archive_write_set_bytes_in_last_block**(), then **archive_write_open_filename**() will adjust the last-block padding depending on the file: it will enable padding when writing to standard output or to a character or block device node, it will disable padding otherwise. You can override this by manually invoking **archive_write_set_bytes_in_last_block**() before calling **archive_write_open**(). The **archive_write_open_filename**() function is safe for use with tape drives or other block-oriented devices.

archive_write_open_memory()
A convenience form of **archive_write_open**() that accepts a pointer to a block of memory that will receive the archive. The final *size_t* * argument points to a variable that will be updated after each write to reflect how much of the buffer is currently in use. You should be careful to ensure that this variable remains allocated until after the archive is closed.

archive_write_header()
Build and write a header using the data in the provided struct archive_entry structure. See archive_entry(3) for information on creating and populating struct archive_entry objects.

archive_write_data()
Write data corresponding to the header just written. Returns number of bytes written or -1 on error.

archive_write_finish_entry()
Close out the entry just written. In particular, this writes out the final padding required by some formats. Ordinarily, clients never need to call this, as it is called automatically by **archive_write_next_header**() and **archive_write_close**() as needed.

archive_write_close()
Complete the archive and invoke the close callback.

archive_write_finish()
Invokes **archive_write_close**() if it was not invoked manually, then releases all resources. Note that this function was declared to return *void* in libarchive 1.x, which made it impossible to detect errors when **archive_write_close**() was invoked implicitly from this function. This is corrected beginning with libarchive 2.0.

More information about the *struct archive* object and the overall design of the library can be found in the libarchive(3) overview.

IMPLEMENTATION

Compression support is built-in to libarchive, which uses zlib and bzlib to handle gzip and bzip2 compression, respectively.

CLIENT CALLBACKS

To use this library, you will need to define and register callback functions that will be invoked to write data to the resulting archive. These functions are registered by calling **archive_write_open**():

typedef int **archive_open_callback**(*struct archive *, void *client_data*)

The open callback is invoked by **archive_write_open**(). It should return **ARCHIVE_OK** if the underlying file or data source is successfully opened. If the open fails, it should call **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL**.

typedef ssize_t **archive_write_callback**(*struct archive *, void *client_data*, *void *buffer*, *size_t length*)

The write callback is invoked whenever the library needs to write raw bytes to the archive. For correct blocking, each call to the write callback function should translate into a single write(2) system call. This is especially critical when writing archives to tape drives. On success, the write callback should return the number of bytes actually written. On error, the callback should invoke **archive_set_error**() to register an error code and message and return -1.

typedef int **archive_close_callback**(*struct archive *, void *client_data*)

The close callback is invoked by archive_close when the archive processing is complete. The callback should return **ARCHIVE_OK** on success. On failure, the callback should invoke **archive_set_error**() to register an error code and message and return **ARCHIVE_FATAL.**

EXAMPLE

The following sketch illustrates basic usage of the library. In this example, the callback functions are simply wrappers around the standard open(2), write(2), and close(2) system calls.

```
#include <sys/stat.h>
#include <archive.h>
#include <archive_entry.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

struct mydata {
        const char *name;
        int fd;
};

int
myopen(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  mydata->fd = open(mydata->name, O_WRONLY | O_CREAT, 0644);
  if (mydata->fd >= 0)
    return (ARCHIVE_OK);
  else
    return (ARCHIVE_FATAL);
}

ssize_t
mywrite(struct archive *a, void *client_data, void *buff, size_t n)
{
  struct mydata *mydata = client_data;

  return (write(mydata->fd, buff, n));
}

int
myclose(struct archive *a, void *client_data)
{
  struct mydata *mydata = client_data;

  if (mydata->fd > 0)
    close(mydata->fd);
  return (0);
}

void
write_archive(const char *outname, const char **filename)
{
  struct mydata *mydata = malloc(sizeof(struct mydata));
  struct archive *a;
  struct archive_entry *entry;
  struct stat st;
  char buff[8192];
  int len;
  int fd;
```

```
  a = archive_write_new();
  mydata->name = outname;
  archive_write_set_compression_gzip(a);
  archive_write_set_format_ustar(a);
  archive_write_open(a, mydata, myopen, mywrite, myclose);
  while (*filename) {
    stat(*filename, &st);
    entry = archive_entry_new();
    archive_entry_copy_stat(entry, &st);
    archive_entry_set_pathname(entry, *filename);
    archive_write_header(a, entry);
    fd = open(*filename, O_RDONLY);
    len = read(fd, buff, sizeof(buff));
    while ( len > 0 ) {
        archive_write_data(a, buff, len);
        len = read(fd, buff, sizeof(buff));
    }
    archive_entry_free(entry);
    filename++;
  }
  archive_write_finish(a);
}

int main(int argc, const char **argv)
{
        const char *outname;
        argv++;
        outname = argv++;
        write_archive(outname, argv);
        return 0;
}
```

RETURN VALUES

Most functions return **ARCHIVE_OK** (zero) on success, or one of several non-zero error codes for errors. Specific error codes include: **ARCHIVE_RETRY** for operations that might succeed if retried, **ARCHIVE_WARN** for unusual conditions that do not prevent further operations, and **ARCHIVE_FATAL** for serious errors that make remaining operations impossible. The **archive_errno**() and **archive_error_string**() functions can be used to retrieve an appropriate error code and a textual error message.

archive_write_new() returns a pointer to a newly-allocated struct archive object.

archive_write_data() returns a count of the number of bytes actually written. On error, -1 is returned and the **archive_errno**() and **archive_error_string**() functions will return appropriate values. Note that if the client-provided write callback function returns a non-zero value, that error will be propagated back to the caller through whatever API function resulted in that call, which may include **archive_write_header**(), **archive_write_data**(), **archive_write_close**(), or **archive_write_finish**(). The client callback can call **archive_set_error**() to provide values that can then be retrieved by **archive_errno**() and **archive_error_string**().

SEE ALSO

tar(1), libarchive(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

There are many peculiar bugs in historic tar implementations that may cause certain programs to reject archives written by this library. For example, several historic implementations calculated header checksums incorrectly and will thus reject valid archives; GNU tar does not fully support pax interchange format; some old tar implementations required specific field terminations.

The default pax interchange format eliminates most of the historic tar limitations and provides a generic key/value attribute facility for vendor-defined extensions. One oversight in POSIX is the failure to provide a standard attribute for large device numbers. This library uses "SCHILY.devminor" and "SCHILY.devmajor" for device numbers that exceed the range supported by the backwards-compatible ustar header. These keys are compatible with Joerg Schilling's **star** archiver. Other implementations may not recognize these keys and will thus be unable to correctly restore device nodes with large device numbers from archives created by this library. archive_write_disk

NAME

archive_write_disk_new, **archive_write_disk_set_options**, **archive_write_disk_set_skip_file**, **archive_write_disk_set_group_lookup**, **archive_write_disk_set_standard_lookup**, **archive_write_disk_set_user_lookup**, **archive_write_header**, **archive_write_data**, **archive_write_finish_entry**, **archive_write_close**, **archive_write_finish** — functions for creating objects on disk

SYNOPSIS

```
#include <archive.h>

struct archive *
archive_write_disk_new(void);

int
archive_write_disk_set_options(struct archive *, int flags);

int
archive_write_disk_set_skip_file(struct archive *, dev_t, ino_t);

int
archive_write_disk_set_group_lookup(struct archive *, void *,
    gid_t (*)(void *, const char *gname, gid_t gid),
    void (*cleanup)(void *));

int
archive_write_disk_set_standard_lookup(struct archive *);

int
archive_write_disk_set_user_lookup(struct archive *, void *,
    uid_t (*)(void *, const char *uname, uid_t uid),
    void (*cleanup)(void *));

int
archive_write_header(struct archive *, struct archive_entry *);

ssize_t
archive_write_data(struct archive *, const void *, size_t);

int
archive_write_finish_entry(struct archive *);

int
archive_write_close(struct archive *);

int
archive_write_finish(struct archive *);
```

DESCRIPTION

These functions provide a complete API for creating objects on disk from struct archive_entry descriptions. They are most naturally used when extracting objects from an archive using the **archive_read**() interface. The general process is to read struct archive_entry objects from an archive, then write those objects to a struct archive object created using the **archive_write_disk**() family functions. This interface is deliberately very similar to the **archive_write**() interface used to write objects to a streaming archive.

archive_write_disk_new()
: Allocates and initializes a struct archive object suitable for writing objects to disk.

archive_write_disk_set_skip_file()
: Records the device and inode numbers of a file that should not be overwritten. This is typically used to ensure that an extraction process does not overwrite the archive from which objects are being read. This capability is technically unnecessary but can be a significant per-

formance optimization in practice.

archive_write_disk_set_options()
The options field consists of a bitwise OR of one or more of the following values:

ARCHIVE_EXTRACT_OWNER
The user and group IDs should be set on the restored file. By default, the user and group IDs are not restored.

ARCHIVE_EXTRACT_PERM
Full permissions (including SGID, SUID, and sticky bits) should be restored exactly as specified, without obeying the current umask. Note that SUID and SGID bits can only be restored if the user and group ID of the object on disk are correct. If **ARCHIVE_EXTRACT_OWNER** is not specified, then SUID and SGID bits will only be restored if the default user and group IDs of newly-created objects on disk happen to match those specified in the archive entry. By default, only basic permissions are restored, and umask is obeyed.

ARCHIVE_EXTRACT_TIME
The timestamps (mtime, ctime, and atime) should be restored. By default, they are ignored. Note that restoring of atime is not currently supported.

ARCHIVE_EXTRACT_NO_OVERWRITE
Existing files on disk will not be overwritten. By default, existing regular files are truncated and overwritten; existing directories will have their permissions updated; other pre-existing objects are unlinked and recreated from scratch.

ARCHIVE_EXTRACT_UNLINK
Existing files on disk will be unlinked before any attempt to create them. In some cases, this can prove to be a significant performance improvement. By default, existing files are truncated and rewritten, but the file is not recreated. In particular, the default behavior does not break existing hard links.

ARCHIVE_EXTRACT_ACL
Attempt to restore ACLs. By default, extended ACLs are ignored.

ARCHIVE_EXTRACT_FFLAGS
Attempt to restore extended file flags. By default, file flags are ignored.

ARCHIVE_EXTRACT_XATTR
Attempt to restore POSIX.1e extended attributes. By default, they are ignored.

ARCHIVE_EXTRACT_SECURE_SYMLINKS
Refuse to extract any object whose final location would be altered by a symlink on disk. This is intended to help guard against a variety of mischief caused by archives that (deliberately or otherwise) extract files outside of the current directory. The default is not to perform this check. If **ARCHIVE_EXTRACT_UNLINK** is specified together with this option, the library will remove any intermediate symlinks it finds and return an error only if such symlink could not be removed.

ARCHIVE_EXTRACT_SECURE_NODOTDOT
Refuse to extract a path that contains a `..` element anywhere within it. The default is to not refuse such paths. Note that paths ending in `..` always cause an error, regardless of this flag.

archive_write_disk_set_group_lookup(),
archive_write_disk_set_user_lookup()
The struct archive_entry objects contain both names and ids that can be used to identify users and groups. These names and ids describe the ownership of the file itself and also appear in ACL lists. By default, the library uses the ids and ignores the names, but this can be overridden by registering user and group lookup functions. To register, you must provide a lookup function which accepts both a name and id and returns a suitable id. You may also provide a void * pointer to a private data structure and a cleanup function for that data. The cleanup function will be invoked when the struct archive object is destroyed.

archive_write_disk_set_standard_lookup()
This convenience function installs a standard set of user and group lookup functions. These functions use getpwnam(3) and getgrnam(3) to convert names to ids, defaulting to the ids if the names cannot be looked up. These functions also implement a simple memory cache to reduce the number of calls to getpwnam(3) and getgrnam(3).

archive_write_header()
Build and write a header using the data in the provided struct archive_entry structure. See archive_entry(3) for information on creating and populating struct archive_entry objects.

archive_write_data()
Write data corresponding to the header just written. Returns number of bytes written or -1 on error.

archive_write_finish_entry()
Close out the entry just written. Ordinarily, clients never need to call this, as it is called automatically by **archive_write_next_header**() and **archive_write_close**() as needed.

archive_write_close()
Set any attributes that could not be set during the initial restore. For example, directory timestamps are not restored initially because restoring a subsequent file would alter that timestamp. Similarly, non-writable directories are initially created with write permissions (so that their contents can be restored). The **archive_write_disk_new** library maintains a list of all such deferred attributes and sets them when this function is invoked.

archive_write_finish()
Invokes **archive_write_close**() if it was not invoked manually, then releases all resources.

More information about the *struct archive* object and the overall design of the library can be found in the libarchive(3) overview. Many of these functions are also documented under archive_write(3).

RETURN VALUES

Most functions return **ARCHIVE_OK** (zero) on success, or one of several non-zero error codes for errors. Specific error codes include: **ARCHIVE_RETRY** for operations that might succeed if retried, **ARCHIVE_WARN** for unusual conditions that do not prevent further operations, and **ARCHIVE_FATAL** for serious errors that make remaining operations impossible. The **archive_errno**() and **archive_error_string**() functions can be used to retrieve an appropriate error code and a textual error message.

archive_write_disk_new() returns a pointer to a newly-allocated struct archive object.

archive_write_data() returns a count of the number of bytes actually written. On error, -1 is returned and the **archive_errno**() and **archive_error_string**() functions will return appropriate values.

SEE ALSO

archive_read(3), archive_write(3), tar(1), libarchive(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3. The **archive_write_disk** interface was added to **libarchive 2.0** and first appeared in FreeBSD 6.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Directories are actually extracted in two distinct phases. Directories are created during **archive_write_header**(), but final permissions are not set until **archive_write_close**(). This separation is necessary to correctly handle borderline cases such as a non-writable directory con-

taining files, but can cause unexpected results. In particular, directory permissions are not fully restored until the archive is closed. If you use `chdir`(2) to change the current directory between calls to **archive_read_extract**() or before calling **archive_read_close**(), you may confuse the permission-setting logic with the result that directory permissions are restored incorrectly.

The library attempts to create objects with filenames longer than **PATH_MAX** by creating prefixes of the full path and changing the current directory. Currently, this logic is limited in scope; the fixup pass does not work correctly for such objects and the symlink security check option disables the support for very long pathnames.

Restoring the path `aa/../bb` does create each intermediate directory. In particular, the directory `aa` is created as well as the final object `bb`. In theory, this can be exploited to create an entire directory heirarchy with a single request. Of course, this does not work if the **ARCHIVE_EXTRACT_NODOTDOT** option is specified.

Implicit directories are always created obeying the current umask. Explicit objects are created obeying the current umask unless **ARCHIVE_EXTRACT_PERM** is specified, in which case they current umask is ignored.

SGID and SUID bits are restored only if the correct user and group could be set. If **ARCHIVE_EXTRACT_OWNER** is not specified, then no attempt is made to set the ownership. In this case, SGID and SUID bits are restored only if the user and group of the final object happen to match those specified in the entry.

The "standard" user-id and group-id lookup functions are not the defaults because `getgrnam`(3) and `getpwnam`(3) are sometimes too large for particular applications. The current design allows the application author to use a more compact implementation when appropriate.

There should be a corresponding **archive_read_disk** interface that walks a directory heirarchy and returns archive entry objects.. LIBARCHIVE

NAME

libarchive — functions for reading and writing streaming archives

LIBRARY

library "libarchive"

OVERVIEW

The **libarchive** library provides a flexible interface for reading and writing streaming archive files such as tar and cpio. The library is inherently stream-oriented; readers serially iterate through the archive, writers serially add things to the archive. In particular, note that there is no built-in support for random access nor for in-place modification.

When reading an archive, the library automatically detects the format and the compression. The library currently has read support for:

- old-style tar archives,
- most variants of the POSIX "ustar" format,
- the POSIX "pax interchange" format,
- GNU-format tar archives,
- most common cpio archive formats,
- ISO9660 CD images (with or without RockRidge extensions),
- Zip archives.

The library automatically detects archives compressed with gzip(1), bzip2(1), or compress(1) and decompresses them transparently.

When writing an archive, you can specify the compression to be used and the format to use. The library can write

- POSIX-standard "ustar" archives,
- POSIX "pax interchange format" archives,
- POSIX octet-oriented cpio archives,
- two different variants of shar archives.

Pax interchange format is an extension of the tar archive format that eliminates essentially all of the limitations of historic tar formats in a standard fashion that is supported by POSIX-compliant pax(1) implementations on many systems as well as several newer implementations of tar(1). Note that the default write format will suppress the pax extended attributes for most entries; explicitly requesting pax format will enable those attributes for all entries.

The read and write APIs are accessed through the **archive_read_XXX**() functions and the **archive_write_XXX**() functions, respectively, and either can be used independently of the other.

The rest of this manual page provides an overview of the library operation. More detailed information can be found in the individual manual pages for each API or utility function.

READING AN ARCHIVE

To read an archive, you must first obtain an initialized struct archive object from **archive_read_new**(). You can then modify this object for the desired operations with the various **archive_read_set_XXX**() and **archive_read_support_XXX**() functions. In particular, you will need to invoke appropriate **archive_read_support_XXX**() functions to enable the corresponding compression and format support. Note that these latter functions perform two distinct operations: they cause the corresponding support code to be linked into your program, and they enable the corresponding auto-detect code. Unless you have specific constraints, you will generally want to invoke **archive_read_support_compression_all**() and **archive_read_support_format_all**() to enable auto-detect for all formats and compression types currently supported by the library.

Once you have prepared the struct archive object, you call **archive_read_open**() to actually open the archive and prepare it for reading. There are several variants of this function; the most basic expects you to provide pointers to several functions that can provide blocks of bytes from the archive. There are convenience forms that allow you to specify a filename, file descriptor, *FILE* * object, or a block of memory from which to read the archive data. Note that the core library makes no assumptions about the size

of the blocks read; callback functions are free to read whatever block size is most appropriate for the medium.

Each archive entry consists of a header followed by a certain amount of data. You can obtain the next header with **archive_read_next_header**(), which returns a pointer to an struct archive_entry structure with information about the current archive element. If the entry is a regular file, then the header will be followed by the file data. You can use **archive_read_data**() (which works much like the read(2) system call) to read this data from the archive. You may prefer to use the higher-level **archive_read_data_skip**(), which reads and discards the data for this entry, **archive_read_data_to_buffer**(), which reads the data into an in-memory buffer, **archive_read_data_to_file**(), which copies the data to the provided file descriptor, or **archive_read_extract**(), which recreates the specified entry on disk and copies data from the archive. In particular, note that **archive_read_extract**() uses the struct archive_entry structure that you provide it, which may differ from the entry just read from the archive. In particular, many applications will want to override the pathname, file permissions, or ownership.

Once you have finished reading data from the archive, you should call **archive_read_close**() to close the archive, then call **archive_read_finish**() to release all resources, including all memory allocated by the library.

The archive_read(3) manual page provides more detailed calling information for this API.

WRITING AN ARCHIVE

You use a similar process to write an archive. The **archive_write_new**() function creates an archive object useful for writing, the various **archive_write_set_XXX**() functions are used to set parameters for writing the archive, and **archive_write_open**() completes the setup and opens the archive for writing.

Individual archive entries are written in a three-step process: You first initialize a struct archive_entry structure with information about the new entry. At a minimum, you should set the pathname of the entry and provide a *struct stat* with a valid *st_mode* field, which specifies the type of object and *st_size* field, which specifies the size of the data portion of the object. The **archive_write_header**() function actually writes the header data to the archive. You can then use **archive_write_data**() to write the actual data.

After all entries have been written, use the **archive_write_finish**() function to release all resources.

The archive_write(3) manual page provides more detailed calling information for this API.

DESCRIPTION

Detailed descriptions of each function are provided by the corresponding manual pages.

All of the functions utilize an opaque struct archive datatype that provides access to the archive contents.

The struct archive_entry structure contains a complete description of a single archive entry. It uses an opaque interface that is fully documented in archive_entry(3).

Users familiar with historic formats should be aware that the newer variants have eliminated most restrictions on the length of textual fields. Clients should not assume that filenames, link names, user names, or group names are limited in length. In particular, pax interchange format can easily accommodate pathnames in arbitrary character sets that exceed *PATH_MAX*.

RETURN VALUES

Most functions return zero on success, non-zero on error. The return value indicates the general severity of the error, ranging from **ARCHIVE_WARN**, which indicates a minor problem that should probably be reported to the user, to **ARCHIVE_FATAL**, which indicates a serious problem that will prevent any further operations on this archive. On error, the **archive_errno**() function can be used to retrieve a numeric error code (see errno(2)). The **archive_error_string**() returns a textual error message suitable for display.

archive_read_new() and **archive_write_new**() return pointers to an allocated and initialized struct archive object.

archive_read_data() and **archive_write_data**() return a count of the number of bytes actually read or written. A value of zero indicates the end of the data for this entry. A negative value indicates an error, in which case the **archive_errno**() and **archive_error_string**() functions can be used to obtain more information.

ENVIRONMENT

There are character set conversions within the archive_entry(3) functions that are impacted by the currently-selected locale.

SEE ALSO

tar(1), archive_entry(3), archive_read(3), archive_util(3), archive_write(3), tar(5)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

Some archive formats support information that is not supported by struct archive_entry. Such information cannot be fully archived or restored using this library. This includes, for example, comments, character sets, or the arbitrary key/value pairs that can appear in pax interchange format archives.

Conversely, of course, not all of the information that can be stored in an struct archive_entry is supported by all formats. For example, cpio formats do not support nanosecond timestamps; old tar formats do not support large device numbers. LIBARCHIVE

NAME

libarchive_internals — description of libarchive internal interfaces

OVERVIEW

The **libarchive** library provides a flexible interface for reading and writing streaming archive files such as tar and cpio. Internally, it follows a modular layered design that should make it easy to add new archive and compression formats.

GENERAL ARCHITECTURE

Externally, libarchive exposes most operations through an opaque, object-style interface. The archive_entry(1) objects store information about a single filesystem object. The rest of the library provides facilities to write archive_entry(1) objects to archive files, read them from archive files, and write them to disk. (There are plans to add a facility to read archive_entry(1) objects from disk as well.)

The read and write APIs each have four layers: a public API layer, a format layer that understands the archive file format, a compression layer, and an I/O layer. The I/O layer is completely exposed to clients who can replace it entirely with their own functions.

In order to provide as much consistency as possible for clients, some public functions are virtualized. Eventually, it should be possible for clients to open an archive or disk writer, and then use a single set of code to select and write entries, regardless of the target.

READ ARCHITECTURE

From the outside, clients use the archive_read(3) API to manipulate an **archive** object to read entries and bodies from an archive stream. Internally, the **archive** object is cast to an **archive_read** object, which holds all read-specific data. The API has four layers: The lowest layer is the I/O layer. This layer can be overridden by clients, but most clients use the packaged I/O callbacks provided, for example, by archive_read_open_memory(3), and archive_read_open_fd(3). The compression layer calls the I/O layer to read bytes and decompresses them for the format layer. The format layer unpacks a stream of uncompressed bytes and creates **archive_entry** objects from the incoming data. The API layer tracks overall state (for example, it prevents clients from reading data before reading a header) and invokes the format and compression layer operations through registered function pointers. In particular, the API layer drives the format-detection process: When opening the archive, it reads an initial block of data and offers it to each registered compression handler. The one with the highest bid is initialized with the first block. Similarly, the format handlers are polled to see which handler is the best for each archive. (Prior to 2.4.0, the format bidders were invoked for each entry, but this design hindered error recovery.)

I/O Layer and Client Callbacks

The read API goes to some lengths to be nice to clients. As a result, there are few restrictions on the behavior of the client callbacks.

The client read callback is expected to provide a block of data on each call. A zero-length return does indicate end of file, but otherwise blocks may be as small as one byte or as large as the entire file. In particular, blocks may be of different sizes.

The client skip callback returns the number of bytes actually skipped, which may be much smaller than the skip requested. The only requirement is that the skip not be larger. In particular, clients are allowed to return zero for any skip that they don't want to handle. The skip callback must never be invoked with a negative value.

Keep in mind that not all clients are reading from disk: clients reading from networks may provide different-sized blocks on every request and cannot skip at all; advanced clients may use mmap(2) to read the entire file into memory at once and return the entire file to libarchive as a single block; other clients may begin asynchronous I/O operations for the next block on each request.

Decompresssion Layer

The decompression layer not only handles decompression, it also buffers data so that the format handlers see a much nicer I/O model. The decompression API is a two stage peek/consume model. A read_ahead request specifies a minimum read amount; the decompression layer must provide a pointer to at least that much data. If more data is immediately available, it should return more: the format layer handles bulk data reads by asking for a minimum of one byte and then copying as much data as is available.

A subsequent call to the **consume**() function advances the read pointer. Note that data returned from a **read_ahead**() call is guaranteed to remain in place until the next call to **read_ahead**(). Intervening calls to **consume**() should not cause the data to move.

Skip requests must always be handled exactly. Decompression handlers that cannot seek forward should not register a skip handler; the API layer fills in a generic skip handler that reads and discards data.

A decompression handler has a specific lifecycle:

Registration/Configuration

When the client invokes the public support function, the decompression handler invokes the internal **__archive_read_register_compression**() function to provide bid and initialization functions. This function returns **NULL** on error or else a pointer to a **struct decompressor_t**. This structure contains a *void * config* slot that can be used for storing any customization information.

Bid The bid function is invoked with a pointer and size of a block of data. The decompressor can access its config data through the *decompressor* element of the **archive_read** object. The bid function is otherwise stateless. In particular, it must not perform any I/O operations.

The value returned by the bid function indicates its suitability for handling this data stream. A bid of zero will ensure that this decompressor is never invoked. Return zero if magic number checks fail. Otherwise, your initial implementation should return the number of bits actually checked. For example, if you verify two full bytes and three bits of another byte, bid 19. Note that the initial block may be very short; be careful to only inspect the data you are given. (The current decompressors require two bytes for correct bidding.)

Initialize The winning bidder will have its init function called. This function should initialize the remaining slots of the *struct decompressor_t* object pointed to by the *decompressor* element of the *archive_read* object. In particular, it should allocate any working data it needs in the *data* slot of that structure. The init function is called with the block of data that was used for tasting. At this point, the decompressor is responsible for all I/O requests to the client callbacks. The decompressor is free to read more data as and when necessary.

Satisfy I/O requests

The format handler will invoke the *read_ahead*, *consume*, and *skip* functions as needed.

Finish The finish method is called only once when the archive is closed. It should release anything stored in the *data* and *config* slots of the *decompressor* object. It should not invoke the client close callback.

Format Layer

The read formats have a similar lifecycle to the decompression handlers:

Registration

Allocate your private data and initialize your pointers.

Bid Formats bid by invoking the **read_ahead**() decompression method but not calling the **consume**() method. This allows each bidder to look ahead in the input stream. Bidders should not look further ahead than necessary, as long look aheads put pressure on the decompression layer to buffer lots of data. Most formats only require a few hundred bytes of look ahead; look aheads of a few kilobytes are reasonable. (The ISO9660 reader sometimes looks ahead by 48k, which should be considered an upper limit.)

Read header

The header read is usually the most complex part of any format. There are a few strategies worth mentioning: For formats such as tar or cpio, reading and parsing the header is straightforward since headers alternate with data. For formats that store all header data at the beginning of the file, the first header read request may have to read all headers into memory and

store that data, sorted by the location of the file data. Subsequent header read requests will skip forward to the beginning of the file data and return the corresponding header.

Read Data
: The read data interface supports sparse files; this requires that each call return a block of data specifying the file offset and size. This may require you to carefully track the location so that you can return accurate file offsets for each read. Remember that the decompressor will return as much data as it has. Generally, you will want to request one byte, examine the return value to see how much data is available, and possibly trim that to the amount you can use. You should invoke consume for each block just before you return it.

Skip All Data
: The skip data call should skip over all file data and trailing padding. This is called automatically by the API layer just before each header read. It is also called in response to the client calling the public **data_skip**() function.

Cleanup
: On cleanup, the format should release all of its allocated memory.

API Layer

XXX to do XXX

WRITE ARCHITECTURE

The write API has a similar set of four layers: an API layer, a format layer, a compression layer, and an I/O layer. The registration here is much simpler because only one format and one compression can be registered at a time.

I/O Layer and Client Callbacks

XXX To be written XXX

Compression Layer

XXX To be written XXX

Format Layer

XXX To be written XXX

API Layer

XXX To be written XXX

WRITE_DISK ARCHITECTURE

The write_disk API is intended to look just like the write API to clients. Since it does not handle multiple formats or compression, it is not layered internally.

GENERAL SERVICES

The **archive_read**, **archive_write**, and **archive_write_disk** objects all contain an initial **archive** object which provides common support for a set of standard services. (Recall that ANSI/ISO C90 guarantees that you can cast freely between a pointer to a structure and a pointer to the first element of that structure.) The **archive** object has a magic value that indicates which API this object is associated with, slots for storing error information, and function pointers for virtualized API functions.

MISCELLANEOUS NOTES

Connecting existing archiving libraries into libarchive is generally quite difficult. In particular, many existing libraries strongly assume that you are reading from a file; they seek forwards and backwards as necessary to locate various pieces of information. In contrast, libarchive never seeks backwards in its input, which sometimes requires very different approaches.

For example, libarchive's ISO9660 support operates very differently from most ISO9660 readers. The libarchive support utilizes a work-queue design that keeps a list of known entries sorted by their location in the input. Whenever libarchive's ISO9660 implementation is asked for the next header, checks this list to find the next item on the disk. Directories are parsed when they are encountered and new items are added to the list. This design relies heavily on the ISO9660 image being optimized so that directo-

ries always occur earlier on the disk than the files they describe.

Depending on the specific format, such approaches may not be possible. The ZIP format specification, for example, allows archivers to store key information only at the end of the file. In theory, it is possible to create ZIP archives that cannot be read without seeking. Fortunately, such archives are very rare, and libarchive can read most ZIP archives, though it cannot always extract as much information as a dedicated ZIP program.

SEE ALSO

`archive`(3), `archive_entry`(3), `archive_read`(3), `archive_write`(3), `archive_write_disk`(3)

HISTORY

The **libarchive** library first appeared in FreeBSD 5.3.

AUTHORS

The **libarchive** library was written by Tim Kientzle ⟨kientzle@acm.org⟩.

BUGS

CPIO

NAME

cpio — format of cpio archive files

DESCRIPTION

The **cpio** archive format collects any number of files, directories, and other file system objects (symbolic links, device nodes, etc.) into a single stream of bytes.

General Format

Each file system object in a **cpio** archive comprises a header record with basic numeric metadata followed by the full pathname of the entry and the file data. The header record stores a series of integer values that generally follow the fields in *struct stat*. (See `stat`(2) for details.) The variants differ primarily in how they store those integers (binary, octal, or hexadecimal). The header is followed by the pathname of the entry (the length of the pathname is stored in the header) and any file data. The end of the archive is indicated by a special record with the pathname "TRAILER!!!".

PWB format

XXX Any documentation of the original PWB/UNIX 1.0 format? XXX

Old Binary Format

The old binary **cpio** format stores numbers as 2-byte and 4-byte binary values. Each entry begins with a header in the following format:

```
struct header_old_cpio {
        unsigned short   c_magic;
        unsigned short   c_dev;
        unsigned short   c_ino;
        unsigned short   c_mode;
        unsigned short   c_uid;
        unsigned short   c_gid;
        unsigned short   c_nlink;
        unsigned short   c_rdev;
        unsigned short   c_mtime[2];
        unsigned short   c_namesize;
        unsigned short   c_filesize[2];
};
```

The *unsigned short* fields here are 16-bit integer values; the *unsigned int* fields are 32-bit integer values. The fields are as follows

magic The integer value octal 070707. This value can be used to determine whether this archive is written with little-endian or big-endian integers.

dev, *ino* The device and inode numbers from the disk. These are used by programs that read **cpio** archives to determine when two entries refer to the same file. Programs that synthesize **cpio** archives should be careful to set these to distinct values for each entry.

mode The mode specifies both the regular permissions and the file type. It consists of several bit fields as follows:

- 0170000 This masks the file type bits.
- 0140000 File type value for sockets.
- 0120000 File type value for symbolic links. For symbolic links, the link body is stored as file data.
- 0100000 File type value for regular files.
- 0060000 File type value for block special devices.
- 0040000 File type value for directories.
- 0020000 File type value for character special devices.

0010000 File type value for named pipes or FIFOs.
0004000 SUID bit.
0002000 SGID bit.
0001000 Sticky bit. On some systems, this modifies the behavior of executables and/or directories.
0000777 The lower 9 bits specify read/write/execute permissions for world, group, and user following standard POSIX conventions.

uid, *gid* The numeric user id and group id of the owner.

nlink The number of links to this file. Directories always have a value of at least two here. Note that hardlinked files include file data with every copy in the archive.

rdev For block special and character special entries, this field contains the associated device number. For all other entry types, it should be set to zero by writers and ignored by readers.

mtime Modification time of the file, indicated as the number of seconds since the start of the epoch, 00:00:00 UTC January 1, 1970. The four-byte integer is stored with the most-significant 16 bits first followed by the least-significant 16 bits. Each of the two 16 bit values are stored in machine-native byte order.

namesize
The number of bytes in the pathname that follows the header. This count includes the trailing NULL byte.

filesize The size of the file. Note that this archive format is limited to four gigabyte file sizes. See *mtime* above for a description of the storage of four-byte integers.

The pathname immediately follows the fixed header. If the **namesize** is odd, an additional NULL byte is added after the pathname. The file data is then appended, padded with NULL bytes to an even length.

Hardlinked files are not given special treatment; the full file contents are included with each copy of the file.

Portable ASCII Format

Version 2 of the Single UNIX Specification ("SUSv2") standardized an ASCII variant that is portable across all platforms. It is commonly known as the "old character" format or as the "odc" format. It stores the same numeric fields as the old binary format, but represents them as 6-character or 11-character octal values.

```
struct cpio_odc_header {
        char    c_magic[6];
        char    c_dev[6];
        char    c_ino[6];
        char    c_mode[6];
        char    c_uid[6];
        char    c_gid[6];
        char    c_nlink[6];
        char    c_rdev[6];
        char    c_mtime[11];
        char    c_namesize[6];
        char    c_filesize[11];
};
```

The fields are identical to those in the old binary format. The name and file body follow the fixed header. Unlike the old binary format, there is no additional padding after the pathname or file contents. If the files being archived are themselves entirely ASCII, then the resulting archive will be entirely ASCII, except for the NULL byte that terminates the name field.

New ASCII Format

The "new" ASCII format uses 8-byte hexadecimal fields for all numbers and separates device numbers into separate fields for major and minor numbers.

```
struct cpio_newc_header {
        char    c_magic[6];
        char    c_ino[8];
        char    c_mode[8];
        char    c_uid[8];
        char    c_gid[8];
        char    c_nlink[8];
        char    c_mtime[8];
        char    c_filesize[8];
        char    c_devmajor[8];
        char    c_devminor[8];
        char    c_rdevmajor[8];
        char    c_rdevminor[8];
        char    c_namesize[8];
        char    c_check[8];
};
```

Except as specified below, the fields here match those specified for the old binary format above.

magic The string “070701”.

check This field is always set to zero by writers and ignored by readers. See the next section for more details.

The pathname is followed by NULL bytes so that the total size of the fixed header plus pathname is a multiple of four. Likewise, the file data is padded to a multiple of four bytes. Note that this format supports only 4 gigabyte files (unlike the older ASCII format, which supports 8 gigabyte files).

In this format, hardlinked files are handled by setting the filesize to zero for each entry except the last one that appears in the archive.

New CRC Format

The CRC format is identical to the new ASCII format described in the previous section except that the magic field is set to “070702” and the *check* field is set to the sum of all bytes in the file data. This sum is computed treating all bytes as unsigned values and using unsigned arithmetic. Only the least-significant 32 bits of the sum are stored.

HP variants

The **cpio** implementation distributed with HPUX used XXXX but stored device numbers differently XXX.

Other Extensions and Variants

Sun Solaris uses additional file types to store extended file data, including ACLs and extended attributes, as special entries in cpio archives.

XXX Others? XXX

BUGS

The “CRC” format is mis-named, as it uses a simple checksum and not a cyclic redundancy check.

The old binary format is limited to 16 bits for user id, group id, device, and inode numbers. It is limited to 4 gigabyte file sizes.

The old ASCII format is limited to 18 bits for the user id, group id, device, and inode numbers. It is limited to 8 gigabyte file sizes.

The new ASCII format is limited to 4 gigabyte file sizes.

None of the cpio formats store user or group names, which are essential when moving files between systems with dissimilar user or group numbering.

Especially when writing older cpio variants, it may be necessary to map actual device/inode values to synthesized values that fit the available fields. With very large filesystems, this may be necessary even for the newer formats.

SEE ALSO

cpio(1), tar(5)

STANDARDS

The **cpio** utility is no longer a part of POSIX or the Single Unix Standard. It last appeared in Version 2 of the Single UNIX Specification ("SUSv2"). It has been supplanted in subsequent standards by pax(1). The portable ASCII format is currently part of the specification for the pax(1) utility.

HISTORY

The original cpio utility was written by Dick Haight while working in AT&T's Unix Support Group. It appeared in 1977 as part of PWB/UNIX 1.0, the "Programmer's Work Bench" derived from Version 6 AT&T UNIX that was used internally at AT&T. Both the old binary and old character formats were in use by 1980, according to the System III source released by SCO under their "Ancient Unix" license. The character format was adopted as part of IEEE Std 1003.1-1988 ("POSIX.1"). XXX when did "newc" appear? Who invented it? When did HP come out with their variant? When did Sun introduce ACLs and extended attributes? XXX. libarchive-formats

NAME

libarchive-formats — archive formats supported by the libarchive library

DESCRIPTION

The libarchive(3) library reads and writes a variety of streaming archive formats. Generally speaking, all of these archive formats consist of a series of "entries". Each entry stores a single file system object, such as a file, directory, or symbolic link.

The following provides a brief description of each format supported by libarchive, with some information about recognized extensions or limitations of the current library support. Note that just because a format is supported by libarchive does not imply that a program that uses libarchive will support that format. Applications that use libarchive specify which formats they wish to support.

Tar Formats

The libarchive(3) library can read most tar archives. However, it only writes POSIX-standard "ustar" and "pax interchange" formats.

All tar formats store each entry in one or more 512-byte records. The first record is used for file metadata, including filename, timestamp, and mode information, and the file data is stored in subsequent records. Later variants have extended this by either appropriating undefined areas of the header record, extending the header to multiple records, or by storing special entries that modify the interpretation of subsequent entries.

gnutar The libarchive(3) library can read GNU-format tar archives. It currently supports the most popular GNU extensions, including modern long filename and linkname support, as well as atime and ctime data. The libarchive library does not support multi-volume archives, nor the old GNU long filename format. It can read GNU sparse file entries, including the new POSIX-based formats, but cannot write GNU sparse file entries.

pax The libarchive(3) library can read and write POSIX-compliant pax interchange format archives. Pax interchange format archives are an extension of the older ustar format that adds a separate entry with additional attributes stored as key/value pairs. The presence of this additional entry is the only difference between pax interchange format and the older ustar format. The extended attributes are of unlimited length and are stored as UTF-8 Unicode strings. Keywords defined in the standard are in all lowercase; vendors are allowed to define custom keys by preceding them with the vendor name in all uppercase. When writing pax archives, libarchive uses many of the SCHILY keys defined by Joerg Schilling's "star" archiver. The libarchive library can read most of the SCHILY keys. It silently ignores any keywords that it does not understand.

restricted pax
The libarchive library can also write pax archives in which it attempts to suppress the extended attributes entry whenever possible. The result will be identical to a ustar archive unless the extended attributes entry is required to store a long file name, long linkname, extended ACL, file flags, or if any of the standard ustar data (user name, group name, UID, GID, etc) cannot be fully represented in the ustar header. In all cases, the result can be dearchived by any program that can read POSIX-compliant pax interchange format archives. Programs that correctly read ustar format (see below) will also be able to read this format; any extended attributes will be extracted as separate files stored in PaxHeader directories.

ustar The libarchive library can both read and write this format. This format has the following limitations:

- Device major and minor numbers are limited to 21 bits. Nodes with larger numbers will not be added to the archive.
- Path names in the archive are limited to 255 bytes. (Shorter if there is no / character in exactly the right place.)

- Symbolic links and hard links are stored in the archive with the name of the referenced file. This name is limited to 100 bytes.
- Extended attributes, file flags, and other extended security information cannot be stored.
- Archive entries are limited to 2 gigabytes in size.

Note that the pax interchange format has none of these restrictions.

The libarchive library can also read a variety of commonly-used extensions to the basic tar format. In particular, it supports base-256 values in certain numeric fields. This essentially removes the limitations on file size, modification time, and device numbers.

The first tar program appeared in Seventh Edition Unix in 1979. The first official standard for the tar file format was the "ustar" (Unix Standard Tar) format defined by POSIX in 1988. POSIX.1-2001 extended the ustar format to create the "pax interchange" format.

Cpio Formats

The libarchive library can read a number of common cpio variants and can write "odc" and "newc" format archives. A cpio archive stores each entry as a fixed-size header followed by a variable-length filename and variable-length data. Unlike tar, cpio does only minimal padding of the header or file data. There are a variety of cpio formats, which differ primarily in how they store the initial header: some store the values as octal or hexadecimal numbers in ASCII, others as binary values of varying byte order and length.

binary The libarchive library can read both big-endian and little-endian variants of the original binary cpio format. This format used 32-bit binary values for file size and mtime, and 16-bit binary values for the other fields.

odc The libarchive library can both read and write this POSIX-standard format. This format stores the header contents as octal values in ASCII. It is standard, portable, and immune from byte-order confusion. File sizes and mtime are limited to 33 bits (8GB file size), other fields are limited to 18 bits.

SVR4 The libarchive library can read both CRC and non-CRC variants of this format. The SVR4 format uses eight-digit hexadecimal values for all header fields. This limits file size to 4GB, and also limits the mtime and other fields to 32 bits. The SVR4 format can optionally include a CRC of the file contents, although libarchive does not currently verify this CRC.

Cpio first appeared in PWB/UNIX 1.0, which was released within AT&T in 1977. PWB/UNIX 1.0 formed the basis of System III Unix, released outside of AT&T in 1981. This makes cpio older than tar, although cpio was not included in Version 7 AT&T Unix. As a result, the tar command became much better known in universities and research groups that used Version 7. The combination of the **find** and **cpio** utilities provided very precise control over file selection. Unfortunately, the format has many limitations that make it unsuitable for widespread use. Only the POSIX format permits files over 4GB, and its 18-bit limit for most other fields makes it unsuitable for modern systems. In addition, cpio formats only store numeric UID/GID values (not usernames and group names), which can make it very difficult to correctly transfer archives across systems with dissimilar user numbering.

Shar Formats

A "shell archive" is a shell script that, when executed on a POSIX-compliant system, will recreate a collection of file system objects. The libarchive library can write two different kinds of shar archives:

shar The traditional shar format uses a limited set of POSIX commands, including `echo`(1), `mkdir`(1), and `sed`(1). It is suitable for portably archiving small collections of plain text files. However, it is not generally well-suited for large archives (many implementations of `sh`(1) have limits on the size of a script) nor should it be used with non-text files.

shardump

This format is similar to shar but encodes files using `uuencode`(1) so that the result will be a plain text file regardless of the file contents. It also includes additional shell commands that attempt to reproduce as many file attributes as possible, including owner, mode, and flags. The additional commands used to restore file attributes make shardump archives less portable than

plain shar archives.

ISO9660 format
Libarchive can read and extract from files containing ISO9660-compliant CDROM images. It also has partial support for Rockridge extensions. In many cases, this can remove the need to burn a physical CDROM. It also avoids security and complexity issues that come with virtual mounts and loopback devices.

Zip format
Libarchive can extract from most zip format archives. It currently only supports uncompressed entries and entries compressed with the "deflate" algorithm. Older zip compression algorithms are not supported.

Archive (library) file format
The Unix archive format (commonly created by the `ar`(1) archiver) is a general-purpose format which is used almost exclusively for object files to be read by the link editor `ld`(1). The ar format has never been standardised. There are two common variants: the GNU format derived from SVR4, and the BSD format, which first appeared in 4.4BSD. Libarchive provides read and write support for both variants.

SEE ALSO
MTREE

NAME

mtree — format of mtree dir heirarchy files

DESCRIPTION

The **mtree** format is a textual format that describes a collection of filesystem objects. Such files are typically used to create or verify directory heirarchies.

General Format

An **mtree** file consists of a series of lines, each providing information about a single filesystem object. Leading whitespace is always ignored.

When encoding file or pathnames, any backslash character or character outside of the 95 printable ASCII characters must be encoded as a a backslash followed by three octal digits. When reading mtree files, any appearance of a backslash followed by three octal digits should be converted into the corresponding character.

Each line is interpreted independently as one of the following types:

Signature — The first line of any mtree file must begin with "#mtree". If a file contains any full path entries, the first line should begin with "#mtree v2.0", otherwise, the first line should begin with "#mtree v1.0".

Blank — Blank lines are ignored.

Comment — Lines beginning with **#** are ignored.

Special — Lines beginning with **/** are special commands that influence the interpretation of later lines.

Relative — If the first whitespace-delimited word has no **/** characters, it is the name of a file in the current directory. Any relative entry that describes a directory changes the current directory.

dot-dot — As a special case, a relative entry with the filename **..** changes the current directory to the parent directory. Options on dot-dot entries are always ignored.

Full — If the first whitespace-delimited word has a **/** character after the first character, it is the pathname of a file relative to the starting directory. There can be multiple full entries describing the same file.

Some tools that process **mtree** files may require that multiple lines describing the same file occur consecutively. It is not permitted for the same file to be mentioned using both a relative and a full file specification.

Special commands

Two special commands are currently defined:

/set — This command defines default values for one or more keywords. It is followed on the same line by one or more whitespace-separated keyword definitions. These definitions apply to all following files that do not specify a value for that keyword.

/unset — This command removes any default value set by a previous **/set** command. It is followed on the same line by one or more keywords separated by whitespace.

Keywords

After the filename, a full or relative entry consists of zero or more whitespace-separated keyword definitions. Each such definitions consists of a key from the following list immediately followed by an '=' sign and a value. Software programs reading mtree files should warn about unrecognized keywords.

Currently supported keywords are as follows:

cksum — The checksum of the file using the default algorithm specified by the cksum(1) utility.

contents The full pathname of a file whose contents should be compared to the contents of this file.

flags The file flags as a symbolic name. See `chflags`(1) for information on these names. If no flags are to be set the string "none" may be used to override the current default.

ignore Ignore any file hierarchy below this file.

gid The file group as a numeric value.

gname The file group as a symbolic name.

md5 The MD5 message digest of the file.

md5digest
A synonym for **md5**.

sha1 The FIPS 160-1 ("SHA-1") message digest of the file.

sha1digest
A synonym for **sha1**.

sha256 The FIPS 180-2 ("SHA-256") message digest of the file.

sha256digest
A synonym for **sha256**.

ripemd160digest
The RIPEMD160 message digest of the file.

rmd160 A synonym for **ripemd160digest**.

rmd160digest
A synonym for **ripemd160digest**.

mode The current file's permissions as a numeric (octal) or symbolic value.

nlink The number of hard links the file is expected to have.

nochange Make sure this file or directory exists but otherwise ignore all attributes.

uid The file owner as a numeric value.

uname The file owner as a symbolic name.

size The size, in bytes, of the file.

link The file the symbolic link is expected to reference.

time The last modification time of the file.

type The type of the file; may be set to any one of the following:

block	block special device
char	character special device
dir	directory
fifo	fifo
file	regular file
link	symbolic link
socket	socket

SEE ALSO

`cksum`(1), `find`(1), `mtree`(8)

BUGS

The FreeBSD implementation of mtree does not currently support the **mtree** 2.0 format. The requirement for a "#mtree" signature line is new and not yet widely implemented.

HISTORY

The **mtree** utility appeared in 4.3BSD−Reno. The MD5 digest capability was added in FreeBSD 2.1, in response to the widespread use of programs which can spoof `cksum`(1). The SHA-1 and RIPEMD160 digests were added in FreeBSD 4.0, as new attacks have demonstrated weaknesses in MD5. The SHA-256 digest was added in FreeBSD 6.0. Support for file flags was added in FreeBSD 4.0, and mostly comes from NetBSD. The “full” entry format was added by NetBSD. TAR

NAME

tar — format of tape archive files

DESCRIPTION

The **tar** archive format collects any number of files, directories, and other file system objects (symbolic links, device nodes, etc.) into a single stream of bytes. The format was originally designed to be used with tape drives that operate with fixed-size blocks, but is widely used as a general packaging mechanism.

General Format

A **tar** archive consists of a series of 512-byte records. Each file system object requires a header record which stores basic metadata (pathname, owner, permissions, etc.) and zero or more records containing any file data. The end of the archive is indicated by two records consisting entirely of zero bytes.

For compatibility with tape drives that use fixed block sizes, programs that read or write tar files always read or write a fixed number of records with each I/O operation. These "blocks" are always a multiple of the record size. The most common block size—and the maximum supported by historic implementations—is 10240 bytes or 20 records. (Note: the terms "block" and "record" here are not entirely standard; this document follows the convention established by John Gilmore in documenting **pdtar**.)

Old-Style Archive Format

The original tar archive format has been extended many times to include additional information that various implementors found necessary. This section describes the variant implemented by the tar command included in Version 7 AT&T UNIX, which is one of the earliest widely-used versions of the tar program.

The header record for an old-style **tar** archive consists of the following:

```
struct header_old_tar {
        char name[100];
        char mode[8];
        char uid[8];
        char gid[8];
        char size[12];
        char mtime[12];
        char checksum[8];
        char linkflag[1];
        char linkname[100];
        char pad[255];
};
```

All unused bytes in the header record are filled with nulls.

name Pathname, stored as a null-terminated string. Early tar implementations only stored regular files (including hardlinks to those files). One common early convention used a trailing "/" character to indicate a directory name, allowing directory permissions and owner information to be archived and restored.

mode File mode, stored as an octal number in ASCII.

uid, *gid* User id and group id of owner, as octal numbers in ASCII.

size Size of file, as octal number in ASCII. For regular files only, this indicates the amount of data that follows the header. In particular, this field was ignored by early tar implementations when extracting hardlinks. Modern writers should always store a zero length for hardlink entries.

mtime Modification time of file, as an octal number in ASCII. This indicates the number of seconds since the start of the epoch, 00:00:00 UTC January 1, 1970. Note that negative values should be avoided here, as they are handled inconsistently.

checksum
: Header checksum, stored as an octal number in ASCII. To compute the checksum, set the checksum field to all spaces, then sum all bytes in the header using unsigned arithmetic. This field should be stored as six octal digits followed by a null and a space character. Note that many early implementations of tar used signed arithmetic for the checksum field, which can cause interoperability problems when transferring archives between systems. Modern robust readers compute the checksum both ways and accept the header if either computation matches.

linkflag, *linkname*
: In order to preserve hardlinks and conserve tape, a file with multiple links is only written to the archive the first time it is encountered. The next time it is encountered, the *linkflag* is set to an ASCII '1' and the *linkname* field holds the first name under which this file appears. (Note that regular files have a null value in the *linkflag* field.)

Early tar implementations varied in how they terminated these fields. The tar command in Version 7 AT&T UNIX used the following conventions (this is also documented in early BSD manpages): the pathname must be null-terminated; the mode, uid, and gid fields must end in a space and a null byte; the size and mtime fields must end in a space; the checksum is terminated by a null and a space. Early implementations filled the numeric fields with leading spaces. This seems to have been common practice until the IEEE Std 1003.1-1988 ("POSIX.1") standard was released. For best portability, modern implementations should fill the numeric fields with leading zeros.

Pre-POSIX Archives

An early draft of IEEE Std 1003.1-1988 ("POSIX.1") served as the basis for John Gilmore's **pdtar** program and many system implementations from the late 1980s and early 1990s. These archives generally follow the POSIX ustar format described below with the following variations:

- The magic value is "ustar " (note the following space). The version field contains a space character followed by a null.
- The numeric fields are generally filled with leading spaces (not leading zeros as recommended in the final standard).
- The prefix field is often not used, limiting pathnames to the 100 characters of old-style archives.

POSIX ustar Archives

IEEE Std 1003.1-1988 ("POSIX.1") defined a standard tar file format to be read and written by compliant implementations of `tar`(1). This format is often called the "ustar" format, after the magic value used in the header. (The name is an acronym for "Unix Standard TAR".) It extends the historic format with new fields:

```
struct header_posix_ustar {
        char name[100];
        char mode[8];
        char uid[8];
        char gid[8];
        char size[12];
        char mtime[12];
        char checksum[8];
        char typeflag[1];
        char linkname[100];
        char magic[6];
        char version[2];
        char uname[32];
        char gname[32];
        char devmajor[8];
        char devminor[8];
        char prefix[155];
        char pad[12];
};
```

typeflag Type of entry. POSIX extended the earlier *linkflag* field with several new type values:

- "0" Regular file. NULL should be treated as a synonym, for compatibility purposes.
- "1" Hard link.
- "2" Symbolic link.
- "3" Character device node.
- "4" Block device node.
- "5" Directory.
- "6" FIFO node.
- "7" Reserved.
- Other A POSIX-compliant implementation must treat any unrecognized typeflag value as a regular file. In particular, writers should ensure that all entries have a valid filename so that they can be restored by readers that do not support the corresponding extension. Uppercase letters "A" through "Z" are reserved for custom extensions. Note that sockets and whiteout entries are not archivable.

It is worth noting that the *size* field, in particular, has different meanings depending on the type. For regular files, of course, it indicates the amount of data following the header. For directories, it may be used to indicate the total size of all files in the directory, for use by operating systems that pre-allocate directory space. For all other types, it should be set to zero by writers and ignored by readers.

magic Contains the magic value "ustar" followed by a NULL byte to indicate that this is a POSIX standard archive. Full compliance requires the uname and gname fields be properly set.

version Version. This should be "00" (two copies of the ASCII digit zero) for POSIX standard archives.

uname, *gname*
User and group names, as null-terminated ASCII strings. These should be used in preference to the uid/gid values when they are set and the corresponding names exist on the system.

devmajor, *devminor*
Major and minor numbers for character device or block device entry.

prefix First part of pathname. If the pathname is too long to fit in the 100 bytes provided by the standard format, it can be split at any / character with the first portion going here. If the prefix field is not empty, the reader will prepend the prefix value and a / character to the regular name field to obtain the full pathname.

Note that all unused bytes must be set to `NULL`.

Field termination is specified slightly differently by POSIX than by previous implementations. The *magic*, *uname*, and *gname* fields must have a trailing `NULL`. The *pathname*, *linkname*, and *prefix* fields must have a trailing `NULL` unless they fill the entire field. (In particular, it is possible to store a 256-character pathname if it happens to have a / as the 156th character.) POSIX requires numeric fields to be zero-padded in the front, and allows them to be terminated with either space or `NULL` characters.

Currently, most tar implementations comply with the ustar format, occasionally extending it by adding new fields to the blank area at the end of the header record.

Pax Interchange Format

There are many attributes that cannot be portably stored in a POSIX ustar archive. IEEE Std 1003.1-2001 ("POSIX.1") defined a "pax interchange format" that uses two new types of entries to hold text-formatted metadata that applies to following entries. Note that a pax interchange format archive is a ustar archive in every respect. The new data is stored in ustar-compatible archive entries that use the "x" or "g" typeflag. In particular, older implementations that do not fully support these extensions will extract the metadata into regular files, where the metadata can be examined as necessary.

An entry in a pax interchange format archive consists of one or two standard ustar entries, each with its own header and data. The first optional entry stores the extended attributes for the following entry. This optional first entry has an "x" typeflag and a size field that indicates the total size of the extended attributes. The extended attributes themselves are stored as a series of text-format lines encoded in the

portable UTF-8 encoding. Each line consists of a decimal number, a space, a key string, an equals sign, a value string, and a new line. The decimal number indicates the length of the entire line, including the initial length field and the trailing newline. An example of such a field is:

```
25 ctime=1084839148.1212\n
```

Keys in all lowercase are standard keys. Vendors can add their own keys by prefixing them with an all uppercase vendor name and a period. Note that, unlike the historic header, numeric values are stored using decimal, not octal. A description of some common keys follows:

atime, **ctime**, **mtime**
: File access, inode change, and modification times. These fields can be negative or include a decimal point and a fractional value.

uname, **uid**, **gname**, **gid**
: User name, group name, and numeric UID and GID values. The user name and group name stored here are encoded in UTF8 and can thus include non-ASCII characters. The UID and GID fields can be of arbitrary length.

linkpath
: The full path of the linked-to file. Note that this is encoded in UTF8 and can thus include non-ASCII characters.

path
: The full pathname of the entry. Note that this is encoded in UTF8 and can thus include non-ASCII characters.

realtime.*, **security.***
: These keys are reserved and may be used for future standardization.

size
: The size of the file. Note that there is no length limit on this field, allowing conforming archives to store files much larger than the historic 8GB limit.

SCHILY.*
: Vendor-specific attributes used by Joerg Schilling's **star** implementation.

SCHILY.acl.access, **SCHILY.acl.default**
: Stores the access and default ACLs as textual strings in a format that is an extension of the format specified by POSIX.1e draft 17. In particular, each user or group access specification can include a fourth colon-separated field with the numeric UID or GID. This allows ACLs to be restored on systems that may not have complete user or group information available (such as when NIS/YP or LDAP services are temporarily unavailable).

SCHILY.devminor, **SCHILY.devmajor**
: The full minor and major numbers for device nodes.

SCHILY.dev, SCHILY.ino, **SCHILY.nlinks**
: The device number, inode number, and link count for the entry. In particular, note that a pax interchange format archive using Joerg Schilling's **SCHILY.*** extensions can store all of the data from *struct stat*.

LIBARCHIVE.xattr.*namespace.key*
: Libarchive stores POSIX.1e-style extended attributes using keys of this form. The *key* value is URL-encoded: All non-ASCII characters and the two special characters "=" and "%" are encoded as "%" followed by two uppercase hexadecimal digits. The value of this key is the extended attribute value encoded in base 64. XXX Detail the base-64 format here XXX

VENDOR.*
: XXX document other vendor-specific extensions XXX

Any values stored in an extended attribute override the corresponding values in the regular tar header. Note that compliant readers should ignore the regular fields when they are overridden. This is important, as existing archivers are known to store non-compliant values in the standard header fields in this situation. There are no limits on length for any of these fields. In particular, numeric fields can be arbitrarily large. All text fields are encoded in UTF8. Compliant writers should store only portable 7-bit ASCII characters in the standard ustar header and use extended attributes whenever a text value contains non-

ASCII characters.

In addition to the **x** entry described above, the pax interchange format also supports a **g** entry. The **g** entry is identical in format, but specifies attributes that serve as defaults for all subsequent archive entries. The **g** entry is not widely used.

Besides the new **x** and **g** entries, the pax interchange format has a few other minor variations from the earlier ustar format. The most troubling one is that hardlinks are permitted to have data following them. This allows readers to restore any hardlink to a file without having to rewind the archive to find an earlier entry. However, it creates complications for robust readers, as it is no longer clear whether or not they should ignore the size field for hardlink entries.

GNU Tar Archives

The GNU tar program started with a pre-POSIX format similar to that described earlier and has extended it using several different mechanisms: It added new fields to the empty space in the header (some of which was later used by POSIX for conflicting purposes); it allowed the header to be continued over multiple records; and it defined new entries that modify following entries (similar in principle to the **x** entry described above, but each GNU special entry is single-purpose, unlike the general-purpose **x** entry). As a result, GNU tar archives are not POSIX compatible, although more lenient POSIX-compliant readers can successfully extract most GNU tar archives.

```
struct header_gnu_tar {
        char name[100];
        char mode[8];
        char uid[8];
        char gid[8];
        char size[12];
        char mtime[12];
        char checksum[8];
        char typeflag[1];
        char linkname[100];
        char magic[6];
        char version[2];
        char uname[32];
        char gname[32];
        char devmajor[8];
        char devminor[8];
        char atime[12];
        char ctime[12];
        char offset[12];
        char longnames[4];
        char unused[1];
        struct {
                char offset[12];
                char numbytes[12];
        } sparse[4];
        char isextended[1];
        char realsize[12];
        char pad[17];
};
```

typeflag GNU tar uses the following special entry types, in addition to those defined by POSIX:

7 GNU tar treats type "7" records identically to type "0" records, except on one obscure RTOS where they are used to indicate the pre-allocation of a contiguous file on disk.

D This indicates a directory entry. Unlike the POSIX-standard "5" typeflag, the header is followed by data records listing the names of files in this directory. Each name is preceded by an ASCII "Y" if the file is stored in this archive or "N" if the file is not stored in this archive. Each name is terminated with a null, and an extra null marks the end of the name list. The purpose of this entry is to support incremental backups; a program restoring from such an archive may wish to delete files on disk that did not exist in the directory when the archive was made.

Note that the "D" typeflag specifically violates POSIX, which requires that unrecognized typeflags be restored as normal files. In this case, restoring the "D" entry as a file could interfere with subsequent creation of the like-named directory.

K The data for this entry is a long linkname for the following regular entry.

L The data for this entry is a long pathname for the following regular entry.

M This is a continuation of the last file on the previous volume. GNU multi-volume archives guarantee that each volume begins with a valid entry header. To ensure this, a file may be split, with part stored at the end of one volume, and part stored at the beginning of the next volume. The "M" typeflag indicates that this entry continues an existing file. Such entries can only occur as the first or second entry in an archive (the latter only if the first entry is a volume label). The *size* field specifies the size of this entry. The *offset* field at bytes 369-380 specifies the offset where this file fragment begins. The *realsize* field specifies the total size of the file (which must equal *size* plus *offset*). When extracting, GNU tar checks that the header file name is the one it is expecting, that the header offset is in the correct sequence, and that the sum of offset and size is equal to realsize. FreeBSD's version of GNU tar does not handle the corner case of an archive's being continued in the middle of a long name or other extension header.

N Type "N" records are no longer generated by GNU tar. They contained a list of files to be renamed or symlinked after extraction; this was originally used to support long names. The contents of this record are a text description of the operations to be done, in the form "Rename %s to %s\n" or "Symlink %s to %s\n"; in either case, both filenames are escaped using K&R C syntax.

S This is a "sparse" regular file. Sparse files are stored as a series of fragments. The header contains a list of fragment offset/length pairs. If more than four such entries are required, the header is extended as necessary with "extra" header extensions (an older format that is no longer used), or "sparse" extensions.

V The *name* field should be interpreted as a tape/volume header name. This entry should generally be ignored on extraction.

magic The magic field holds the five characters "ustar" followed by a space. Note that POSIX ustar archives have a trailing null.

version The version field holds a space character followed by a null. Note that POSIX ustar archives use two copies of the ASCII digit "0".

atime, *ctime*
The time the file was last accessed and the time of last change of file information, stored in octal as with *mtime*.

longnames
This field is apparently no longer used.

Sparse *offset / numbytes*
Each such structure specifies a single fragment of a sparse file. The two fields store values as octal numbers. The fragments are each padded to a multiple of 512 bytes in the archive. On extraction, the list of fragments is collected from the header (including any extension headers), and the data is then read and written to the file at appropriate offsets.

isextended

If this is set to non-zero, the header will be followed by additional "sparse header" records. Each such record contains information about as many as 21 additional sparse blocks as shown here:

```
struct gnu_sparse_header {
        struct {
                char offset[12];
                char numbytes[12];
        } sparse[21];
        char    isextended[1];
        char    padding[7];
};
```

realsize A binary representation of the file's complete size, with a much larger range than the POSIX file size. In particular, with **M** type files, the current entry is only a portion of the file. In that case, the POSIX size field will indicate the size of this entry; the *realsize* field will indicate the total size of the file.

Solaris Tar

XXX More Details Needed XXX

Solaris tar (beginning with SunOS XXX 5.7 ?? XXX) supports an "extended" format that is fundamentally similar to pax interchange format, with the following differences:

- Extended attributes are stored in an entry whose type is **X**, not **x**, as used by pax interchange format. The detailed format of this entry appears to be the same as detailed above for the **x** entry.
- An additional **A** entry is used to store an ACL for the following regular entry. The body of this entry contains a seven-digit octal number (whose value is 01000000 plus the number of ACL entries) followed by a zero byte, followed by the textual ACL description.

Other Extensions

One common extension, utilized by GNU tar, star, and other newer **tar** implementations, permits binary numbers in the standard numeric fields. This is flagged by setting the high bit of the first character. This permits 95-bit values for the length and time fields and 63-bit values for the uid, gid, and device numbers. GNU tar supports this extension for the length, mtime, ctime, and atime fields. Joerg Schilling's star program supports this extension for all numeric fields. Note that this extension is largely obsoleted by the extended attribute record prov