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 maximum block size supported by early implementations was 10240 bytes or 20 records. This is still the default for most implementations although block sizes of 1MiB (2048 records) or larger are commonly used with modern high-speed tape drives. (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 seems to be the earliest widely-used version 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 consists of the five characters "ustar" followed by a 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. NUL 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 NUL 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.

name, *prefix*
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 into the prefix field. 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. The standard does not require a trailing / character on directory names, though most implementations still include this for compatibility reasons.

Note that all unused bytes must be set to NUL.

Field termination is specified slightly differently by POSIX than by previous implementations. The *magic*, *uname*, and *gname* fields must have a trailing NUL. The *pathname*, *linkname*, and *prefix* fields must have a trailing NUL 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 requires them to be terminated with either space or NUL 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.

Numeric Extensions

There have been several attempts to extend the range of sizes or times supported by modifying how numbers are stored in the header.

One obvious extension to increase the size of files is to eliminate the terminating characters from the various numeric fields. For example, the standard only allows the size field to contain 11 octal digits, reserving the twelfth byte for a trailing NUL character. Allowing 12 octal digits allows file sizes up to 64 GB.

Another 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 byte. The remainder of the field is treated as a signed twos-complement value. This permits 95-bit values for the length and time fields and 63-bit values for the uid, gid, and device numbers. In particular, this provides a consistent way to handle negative time values. GNU tar supports this extension for the length, mtime, ctime, and atime fields. Joerg Schilling's star program and the libarchive library support this extension for all numeric fields. Note that this extension is largely obsoleted by the extended attribute record provided by the pax interchange format.

Another early GNU extension allowed base-64 values rather than octal. This extension was short-lived and is no longer supported by any implementation.

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.

hdrcharset
: The character set used by the pax extension values. By default, all textual values in the pax extended attributes are assumed to be in UTF-8, including pathnames, user names, and group names. In some cases, it is not possible to translate local conventions into UTF-8. If this key is present and the value is the six-character ASCII string "BINARY", then all textual values are assumed to be in a platform-dependent multi-byte encoding. Note that there are only two valid values for this key: "BINARY" or "ISO-IR 10646 2000 UTF-8". No other values are permitted by the standard, and the latter value should generally not be used as it is the default when this key is not specified. In particular, this flag should not be used as a general mechanism to allow filenames to be stored in arbitrary encodings.

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.fflags
: The file flags.

SCHILY.realsize
: The full size of the file on disk. XXX explain? XXX

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.*
: Vendor-specific attributes used by the **libarchive** library and programs that use it.

LIBARCHIVE.creationtime
: The time when the file was created. (This should not be confused with the POSIX "ctime" attribute, which refers to the time when the file metadata was last changed.)

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.

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. Due to security concerns, "N" records are now generally ignored when reading archives.

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.

GNU tar pax archives

GNU tar 1.14 (XXX check this XXX) and later will write pax interchange format archives when you specify the **--posix** flag. This format follows the pax interchange format closely, using some **SCHILY** tags and introducing new keywords to store sparse file information. There have been three iterations of the sparse file support, referred to as "0.0", "0.1", and "1.0".

GNU.sparse.numblocks, **GNU.sparse.offset**, **GNU.sparse.numbytes**, **GNU.sparse.size**
: The "0.0" format used an initial **GNU.sparse.numblocks** attribute to indicate the number of blocks in the file, a pair of **GNU.sparse.offset** and **GNU.sparse.numbytes** to indicate the offset and size of each block, and a single **GNU.sparse.size** to indicate the full size of the file. This is not the same as the size in the tar header because the latter value does not include the size of any holes. This format required that the order of attributes be preserved and relied on readers accepting multiple appearances of the same attribute names, which is not officially permitted by the standards.

GNU.sparse.map
: The "0.1" format used a single attribute that stored a comma-separated list of decimal numbers. Each pair of numbers indicated the offset and size, respectively, of a block of data. This does not work well if the archive is extracted by an archiver that does not recognize this extension, since many pax implementations simply discard unrecognized attributes.

GNU.sparse.major, **GNU.sparse.minor**, **GNU.sparse.name**, **GNU.sparse.realsize**
: The "1.0" format stores the sparse block map in one or more 512-byte blocks prepended to the file data in the entry body. The pax attributes indicate the existence of this map (via the

GNU.sparse.major and **GNU.sparse.minor** fields) and the full size of the file. The **GNU.sparse.name** holds the true name of the file. To avoid confusion, the name stored in the regular tar header is a modified name so that extraction errors will be apparent to users.

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** header is used to store an ACL for the following regular entry. The body of this entry contains a seven-digit octal number followed by a zero byte, followed by the textual ACL description. The octal value is the number of ACL entries plus a constant that indicates the ACL type: 01000000 for POSIX.1e ACLs and 03000000 for NFSv4 ACLs.

AIX Tar

XXX More details needed XXX

AIX Tar uses a ustar-formatted header with the type **A** for storing coded ACL information. Unlike the Solaris format, AIX tar writes this header after the regular file body to which it applies. The pathname in this header is either **NFS4** or **AIXC** to indicate the type of ACL stored. The actual ACL is stored in platform-specific binary format.

Mac OS X Tar

The tar distributed with Apple's Mac OS X stores most regular files as two separate files in the tar archive. The two files have the same name except that the first one has "._" prepended to the last path element. This special file stores an AppleDouble-encoded binary blob with additional metadata about the second file, including ACL, extended attributes, and resources. To recreate the original file on disk, each separate file can be extracted and the Mac OS X **copyfile**() function can be used to unpack the separate metadata file and apply it to th regular file. Conversely, the same function provides a "pack" option to encode the extended metadata from a file into a separate file whose contents can then be put into a tar archive.

Note that the Apple extended attributes interact badly with long filenames. Since each file is stored with the full name, a separate set of extensions needs to be included in the archive for each one, doubling the overhead required for files with long names.

Summary of tar type codes

The following list is a condensed summary of the type codes used in tar header records generated by different tar implementations. More details about specific implementations can be found above:

NUL
: Early tar programs stored a zero byte for regular files.

0 POSIX standard type code for a regular file.

1 POSIX standard type code for a hard link description.

2 POSIX standard type code for a symbolic link description.

3 POSIX standard type code for a character device node.

4 POSIX standard type code for a block device node.

5 POSIX standard type code for a directory.

6 POSIX standard type code for a FIFO.

7 POSIX reserved.

7 GNU tar used for pre-allocated files on some systems.

A Solaris tar ACL description stored prior to a regular file header.
A AIX tar ACL description stored after the file body.
D GNU tar directory dump.
K GNU tar long linkname for the following header.
L GNU tar long pathname for the following header.
M GNU tar multivolume marker, indicating the file is a continuation of a file from the previous volume.
N GNU tar long filename support. Deprecated.
S GNU tar sparse regular file.
V GNU tar tape/volume header name.
X Solaris tar general-purpose extension header.
g POSIX pax interchange format global extensions.
x POSIX pax interchange format per-file extensions.

SEE ALSO

ar(1), pax(1), tar(1)

STANDARDS

The **tar** 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 ustar format is currently part of the specification for the pax(1) utility. The pax interchange file format is new with IEEE Std 1003.1-2001 ("POSIX.1").

HISTORY

A **tar** command appeared in Seventh Edition Unix, which was released in January, 1979. It replaced the **tp** program from Fourth Edition Unix which in turn replaced the **tap** program from First Edition Unix. John Gilmore's **pdtar** public-domain implementation (circa 1987) was highly influential and formed the basis of **GNU tar** (circa 1988). Joerg Shilling's **star** archiver is another open-source (GPL) archiver (originally developed circa 1985) which features complete support for pax interchange format.

This documentation was written as part of the **libarchive** and **bsdtar** project by Tim Kientzle ⟨kientzle@FreeBSD.org⟩.