

/* List a tar archive, with support routines for reading a tar archive.

Copyright (C) 1988, 1992, 1993, 1994, 1996, 1997, 1998, 1999, 2000,
2001, 2003, 2004, 2005, 2006, 2007 Free Software Foundation, Inc.

Written by John Gilmore, on 1985-08-26.

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the
Free Software Foundation; either version 3, or (at your option) any later
version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA. */

```
#include <system.h>
#include <inttostr.h>
#include <quotearg.h>
```

```
#include "common.h"
```

```
#define max(a, b) ((a) < (b) ? (b) : (a))
```

```
union block *current_header; /* points to current archive header */
enum archive_format current_format; /* recognized format */
union block *recent_long_name; /* recent long name header and contents */
union block *recent_long_link; /* likewise, for long link */
size_t recent_long_name_blocks; /* number of blocks in recent_long_name */
size_t recent_long_link_blocks; /* likewise, for long link */
```

```
static uintmax_t from_header (const char *, size_t, const char *,
                             uintmax_t, uintmax_t, bool, bool);
```

```
/* Base 64 digits; see Internet RFC 2045 Table 1. */
```

```
static char const base_64_digits[64] =
{
  'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
  'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
  'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
  'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
  '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '/'
};
```

```
/* Table of base-64 digit values indexed by unsigned chars.
```

```
The value is 64 for unsigned chars that are not base-64 digits. */
static char base64_map[UCHAR_MAX + 1];
```

```

static void
base64_init (void)
{
    int i;
    memset (base64_map, 64, sizeof base64_map);
    for (i = 0; i < 64; i++)
        base64_map[(int) base_64_digits[i]] = i;
}

/* Main loop for reading an archive. */
void
read_and (void (*do_something) (void))
{
    enum read_header status = HEADER_STILL_UNREAD;
    enum read_header prev_status;
    struct timespec mtime;

    base64_init ();
    name_gather ();

    open_archive (ACCESS_READ);
    do
    {
        prev_status = status;
        tar_stat_destroy (&current_stat_info);

        status = read_header (false);
        switch (status)
        {
            case HEADER_STILL_UNREAD:
            case HEADER_SUCCESS_EXTENDED:
                abort ();

            case HEADER_SUCCESS:

                /* Valid header. We should decode next field (mode) first.
                 * Ensure incoming names are null terminated. */

                if (! name_match (current_stat_info.file_name)
                    || (NEWER_OPTION_INITIALIZED (newer_mtime_option)
                        /* FIXME: We get mtime now, and again later; this causes
                         * duplicate diagnostics if header.mtime is bogus. */
                        && ((mtime.tv_sec
                            = TIME_FROM_HEADER (current_header->header.mtime)),
                            /* FIXME: Grab fractional time stamps from
                             * extended header. */
                            mtime.tv_nsec = 0,
                            current_stat_info.mtime = mtime,
                            OLDER_TAR_STAT_TIME (current_stat_info, m)))
                    || excluded_name (current_stat_info.file_name))
                {

```

s

```

switch (current_header->header.typeflag)
{
case GNUTYPE_VOLHDR:
case GNUTYPE_MULTIVOL:
    break;

case DIRTYTYPE:
    if (show_omitted_dirs_option)
        WARN ((0, 0, _("%s: Omitting"),
            quotearg_colon (current_stat_info.file_name)));
    /* Fall through. */
default:
    decode_header (current_header,
        &current_stat_info, &current_format, 0);
    skip_member ();
    continue;
}
}

(*do_something) ();
continue;

case HEADER_ZERO_BLOCK:
if (block_number_option)
{
    char buf[UINTMAX_STRSIZE_BOUND];
    fprintf (stdlis, _("block %s: ** Block of NULs **\n"),
        STRINGIFY_BIGINT (current_block_ordinal (), buf));
}

set_next_block_after (current_header);

if (!ignore_zeros_option)
{
    char buf[UINTMAX_STRSIZE_BOUND];

    status = read_header (false);
    if (status == HEADER_ZERO_BLOCK)
        break;
    WARN ((0, 0, _("A lone zero block at %s"),
        STRINGIFY_BIGINT (current_block_ordinal (), buf)));
    break;
}
status = prev_status;
continue;

case HEADER_END_OF_FILE:
if (block_number_option)
{
    char buf[UINTMAX_STRSIZE_BOUND];
    fprintf (stdlis, _("block %s: ** End of File **\n"),
        STRINGIFY_BIGINT (current_block_ordinal (), buf));
}
}

```

```

    }
    break;

case HEADER_FAILURE:
    /* If the previous header was good, tell them that we are
       skipping bad ones. */
    set_next_block_after (current_header);
    switch (prev_status)
    {
    case HEADER_STILL_UNREAD:
        ERROR ((0, 0, _("This does not look like a tar archive")));
        /* Fall through. */

    case HEADER_ZERO_BLOCK:
    case HEADER_SUCCESS:
        if (block_number_option)
            {
            char buf[UINTMAX_STRSIZE_BOUND];
            off_t block_ordinal = current_block_ordinal ();
            block_ordinal -= recent_long_name_blocks;
            block_ordinal -= recent_long_link_blocks;
            fprintf (stdlis, _("block %s: "),
                    STRINGIFY_BIGINT (block_ordinal, buf));
            }
        ERROR ((0, 0, _("Skipping to next header")));
        break;

    case HEADER_END_OF_FILE:
    case HEADER_FAILURE:
        /* We are in the middle of a cascade of errors. */
        break;

    case HEADER_SUCCESS_EXTENDED:
        abort ();
    }
    continue;
}
break;
}
while (!all_names_found (&current_stat_info));

close_archive ();
names_notfound ();      /* print names not found */
}

/* Print a header block, based on tar options. */
void
list_archive (void)
{
    off_t block_ordinal = current_block_ordinal ();
    /* Print the header block. */

```

```

decode_header (current_header, &current_stat_info, &current_format, 0);
if (verbose_option)
    print_header (&current_stat_info, block_ordinal);

if (incremental_option)
{
    if (verbose_option > 2)
        {
            if (is_dumpdir (&current_stat_info))
                list_dumpdir (current_stat_info.dumpdir,
                    dumpdir_size (current_stat_info.dumpdir));
        }
}

skip_member ();
}

```

```

/* Check header checksum */
/* The standard BSD tar sources create the checksum by adding up the
bytes in the header as type char. I think the type char was unsigned
on the PDP-11, but it's signed on the Next and Sun. It looks like the
sources to BSD tar were never changed to compute the checksum
correctly, so both the Sun and Next add the bytes of the header as
signed chars. This doesn't cause a problem until you get a file with
a name containing characters with the high bit set. So tar_checksum
computes two checksums -- signed and unsigned. */

```

```

enum read_header
tar_checksum (union block *header, bool silent)
{
    size_t i;
    int unsigned_sum = 0;          /* the POSIX one :-) */
    int signed_sum = 0;          /* the Sun one :( */
    int recorded_sum;
    uintmax_t parsed_sum;
    char *p;

    p = header->buffer;
    for (i = sizeof *header; i-- != 0;)
        {
            unsigned_sum += (unsigned char) *p;
            signed_sum += (signed char) (*p++);
        }

    if (unsigned_sum == 0)
        return HEADER_ZERO_BLOCK;

    /* Adjust checksum to count the "chksum" field as blanks. */

    for (i = sizeof header->header.chksum; i-- != 0;)
        {
            unsigned_sum -= (unsigned char) header->header.chksum[i];
        }
}

```

```

    signed_sum -= (signed char) (header->header.chksum[i]);
}
unsigned_sum += ' ' * sizeof header->header.chksum;
signed_sum += ' ' * sizeof header->header.chksum;

parsed_sum = from_header (header->header.chksum,
                          sizeof header->header.chksum, 0,
                          (uintmax_t) 0,
                          (uintmax_t) TYPE_MAXIMUM (int), true, silent);
if (parsed_sum == (uintmax_t) -1)
    return HEADER_FAILURE;

recorded_sum = parsed_sum;

if (unsigned_sum != recorded_sum && signed_sum != recorded_sum)
    return HEADER_FAILURE;

return HEADER_SUCCESS;
}

/* Read a block that's supposed to be a header block. Return its
   address in "current_header", and if it is good, the file's size
   and names (file name, link name) in *info.

   Return 1 for success, 0 if the checksum is bad, EOF on eof, 2 for a
   block full of zeros (EOF marker).

   If RAW_EXTENDED_HEADERS is nonzero, do not automatically fold the
   GNU long name and link headers into later headers.

   You must always set_next_block_after(current_header) to skip past
   the header which this routine reads. */

enum read_header
read_header_primitive (bool raw_extended_headers, struct tar_stat_info *info)
{
    union block *header;
    union block *header_copy;
    char *bp;
    union block *data_block;
    size_t size, written;
    union block *next_long_name = 0;
    union block *next_long_link = 0;
    size_t next_long_name_blocks;
    size_t next_long_link_blocks;

    while (1)
    {
        enum read_header status;

        header = find_next_block ();
        current_header = header;

```

```

if (!header)
    return HEADER_END_OF_FILE;

if ((status = tar_checksum (header, false)) != HEADER_SUCCESS)
    return status;

/* Good block. Decode file size and return. */

if (header->header.typeflag == LNKTYPE)
    info->stat.st_size = 0; /* links 0 size on tape */
else
    info->stat.st_size = OFF_FROM_HEADER (header->header.size);

if (header->header.typeflag == GNUTYPE_LONGNAME
    || header->header.typeflag == GNUTYPE_LONGLINK
    || header->header.typeflag == XHDTYPE
    || header->header.typeflag == XGLTYPE
    || header->header.typeflag == SOLARIS_XHDTYPE)
{
    if (raw_extended_headers)
        return HEADER_SUCCESS_EXTENDED;
    else if (header->header.typeflag == GNUTYPE_LONGNAME
        || header->header.typeflag == GNUTYPE_LONGLINK)
    {
        size_t name_size = info->stat.st_size;
        size_t n = name_size % BLOCKSIZE;
        size = name_size + BLOCKSIZE;
        if (n)
            size += BLOCKSIZE - n;

        if (name_size != info->stat.st_size || size < name_size)
            xalloc_die ();

        header_copy = xmalloc (size + 1);

        if (header->header.typeflag == GNUTYPE_LONGNAME)
        {
            if (next_long_name)
                free (next_long_name);
            next_long_name = header_copy;
            next_long_name_blocks = size / BLOCKSIZE;
        }
        else
        {
            if (next_long_link)
                free (next_long_link);
            next_long_link = header_copy;
            next_long_link_blocks = size / BLOCKSIZE;
        }

        set_next_block_after (header);
        *header_copy = *header;
    }
}

```

```

bp = header_copy->buffer + BLOCKSIZE;

for (size -= BLOCKSIZE; size > 0; size -= written)
{
    data_block = find_next_block ();
    if (! data_block)
    {
        ERROR ((0, 0, _("Unexpected EOF in archive")));
        break;
    }
    written = available_space_after (data_block);
    if (written > size)
        written = size;

    memcpy (bp, data_block->buffer, written);
    bp += written;
    set_next_block_after ((union block *)
                          (data_block->buffer + written - 1));
}

*bp = '\0';
}
else if (header->header.typeflag == XHDTYPE
        || header->header.typeflag == SOLARIS_XHDTYPE)
    xheader_read (&info->xhdr, header,
                 OFF_FROM_HEADER (header->header.size));
else if (header->header.typeflag == XGLTYPE)
{
    struct xheader xhdr;
    memset (&xhdr, 0, sizeof xhdr);
    xheader_read (&xhdr, header,
                 OFF_FROM_HEADER (header->header.size));
    xheader_decode_global (&xhdr);
    xheader_destroy (&xhdr);
}

/* Loop! */

}
else
{
    char const *name;
    struct posix_header const *h = &current_header->header;
    char namebuf[sizeof h->prefix + 1 + NAME_FIELD_SIZE + 1];

    if (recent_long_name)
        free (recent_long_name);

    if (next_long_name)
    {
        name = next_long_name->buffer + BLOCKSIZE;
        recent_long_name = next_long_name;
    }
}

```

```

    recent_long_name_blocks = next_long_name_blocks;
}
else
{
    /* Accept file names as specified by POSIX.1-1996
    section 10.1.1. */
    char *np = namebuf;

    if (h->prefix[0] && strcmp (h->magic, TMAGIC) == 0)
    {
        memcpy (np, h->prefix, sizeof h->prefix);
        np[sizeof h->prefix] = '\0';
        np += strlen (np);
        *np++ = '/';
    }
    memcpy (np, h->name, sizeof h->name);
    np[sizeof h->name] = '\0';
    name = namebuf;
    recent_long_name = 0;
    recent_long_name_blocks = 0;
}
assign_string (&info->orig_file_name, name);
assign_string (&info->file_name, name);
info->had_trailing_slash = strip_trailing_slashes (info->file_name);

if (recent_long_link)
    free (recent_long_link);

if (next_long_link)
{
    name = next_long_link->buffer + BLOCKSIZE;
    recent_long_link = next_long_link;
    recent_long_link_blocks = next_long_link_blocks;
}
else
{
    memcpy (namebuf, h->linkname, sizeof h->linkname);
    namebuf[sizeof h->linkname] = '\0';
    name = namebuf;
    recent_long_link = 0;
    recent_long_link_blocks = 0;
}
assign_string (&info->link_name, name);

return HEADER_SUCCESS;
}
}
}

```

```

enum read_header
read_header (bool raw_extended_headers)
{

```

```

return read_header_primitive (raw_extended_headers, &current_stat_info);
}

static char *
decode_xform (char *file_name, void *data)
{
    xform_type type = *(xform_type*)data;

    switch (type)
    {
    case xform_symlink:
        /* FIXME: It is not quite clear how and to which extent are the symbolic
           links subject to filename transformation. In the absence of another
           solution, symbolic links are exempt from component stripping and
           name suffix normalization, but subject to filename transformation
           proper. */
        return file_name;

    case xform_link:
        file_name = safer_name_suffix (file_name, true, absolute_names_option);
        break;

    case xform_regfile:
        file_name = safer_name_suffix (file_name, false, absolute_names_option);
        break;
    }

    if (strip_name_components)
    {
        size_t prefix_len = stripped_prefix_len (file_name,
                                                strip_name_components);

        if (prefix_len == (size_t) -1)
            prefix_len = strlen (file_name);
        file_name += prefix_len;
    }
    return file_name;
}

bool
transform_member_name (char **pinput, xform_type type)
{
    return transform_name_fp (pinput, decode_xform, &type);
}

#define ISOCTAL(c) ((c)>='0'&&(c)<='7')

/* Decode things from a file HEADER block into STAT_INFO, also setting
   *FORMAT_POINTER depending on the header block format. If
   DO_USER_GROUP, decode the user/group information (this is useful
   for extraction, but waste time when merely listing).

   read_header() has already decoded the checksum and length, so we don't.

```

This routine should **not** be called twice for the same block, since the two calls might use different `DO_USER_GROUP` values and thus might end up with different uid/gid for the two calls. If anybody wants the uid/gid they should decode it first, and other callers should decode it without uid/gid before calling a routine, e.g. `print_header`, that assumes decoded data. **/*

```
void
decode_header (union block *header, struct tar_stat_info *stat_info,
              enum archive_format *format_pointer, int do_user_group)
{
    enum archive_format format;

    if (strcmp (header->header.magic, TMAGIC) == 0)
    {
        if (header->star_header.prefix[130] == 0
            && ISOCTAL (header->star_header.atime[0])
            && header->star_header.atime[11] == ' '
            && ISOCTAL (header->star_header.ctime[0])
            && header->star_header.ctime[11] == ' ')
            format = STAR_FORMAT;
        else if (stat_info->xhdr.size)
            format = POSIX_FORMAT;
        else
            format = USTAR_FORMAT;
    }
    else if (strcmp (header->header.magic, OLDBGNU_MAGIC) == 0)
        format = OLDBGNU_FORMAT;
    else
        format = V7_FORMAT;
    *format_pointer = format;

    stat_info->stat.st_mode = MODE_FROM_HEADER (header->header.mode);
    stat_info->mtime.tv_sec = TIME_FROM_HEADER (header->header.mtime);
    stat_info->mtime.tv_nsec = 0;
    assign_string (&stat_info->uname,
                  header->header.uname[0] ? header->header.uname : NULL);
    assign_string (&stat_info->gname,
                  header->header.gname[0] ? header->header.gname : NULL);

    if (format == OLDBGNU_FORMAT && incremental_option)
    {
        stat_info->atime.tv_sec = TIME_FROM_HEADER (header->oldgnu_header.atime);
        stat_info->ctime.tv_sec = TIME_FROM_HEADER (header->oldgnu_header.ctime);
        stat_info->atime.tv_nsec = stat_info->ctime.tv_nsec = 0;
    }
    else if (format == STAR_FORMAT)
    {
        stat_info->atime.tv_sec = TIME_FROM_HEADER (header->star_header.atime);
        stat_info->ctime.tv_sec = TIME_FROM_HEADER (header->star_header.ctime);
        stat_info->atime.tv_nsec = stat_info->ctime.tv_nsec = 0;
    }
}
```

```

else
    stat_info->atime = stat_info->ctime = start_time;

if (format == V7_FORMAT)
{
    stat_info->stat.st_uid = UID_FROM_HEADER (header->header.uid);
    stat_info->stat.st_gid = GID_FROM_HEADER (header->header.gid);
    stat_info->stat.st_rdev = 0;
}
else
{
    if (do_user_group)
    {
        /* FIXME: Decide if this should somewhat depend on -p. */

        if (numeric_owner_option
            || !*header->header.uname
            || !uname_to_uid (header->header.uname, &stat_info->stat.st_uid))
            stat_info->stat.st_uid = UID_FROM_HEADER (header->header.uid);

        if (numeric_owner_option
            || !*header->header.gname
            || !gname_to_gid (header->header.gname, &stat_info->stat.st_gid))
            stat_info->stat.st_gid = GID_FROM_HEADER (header->header.gid);
    }

    switch (header->header.typeflag)
    {
        case BLKTYPE:
        case CHRTYPE:
            stat_info->stat.st_rdev =
                makedev (MAJOR_FROM_HEADER (header->header.devmajor),
                        MINOR_FROM_HEADER (header->header.devminor));
            break;

        default:
            stat_info->stat.st_rdev = 0;
    }
}

stat_info->archive_file_size = stat_info->stat.st_size;
xheader_decode (stat_info);

if (sparse_member_p (stat_info))
{
    sparse_fixup_header (stat_info);
    stat_info->is_sparse = true;
}
else
{
    stat_info->is_sparse = false;
    if (((current_format == GNU_FORMAT

```

```

        || current_format == OLDGNU_FORMAT)
        && current_header->header.typeflag == GNUTYPE_DUMPDIR)
    || stat_info->dumpdir)
    stat_info->is_dumpdir = true;
}

transform_member_name (&stat_info->file_name, xform_regfile);
}

/* Convert buffer at WHERE0 of size DIGS from external format to
   uintmax_t. DIGS must be positive. If TYPE is nonnull, the data
   are of type TYPE. The buffer must represent a value in the range
   -MINUS_MINVAL through MAXVAL. If OCTAL_ONLY, allow only octal
   numbers instead of the other GNU extensions. Return -1 on error,
   diagnosing the error if TYPE is nonnull and if !SILENT. */
static uintmax_t
from_header (char const *where0, size_t digs, char const *type,
             uintmax_t minus_minval, uintmax_t maxval,
             bool octal_only, bool silent)
{
    uintmax_t value;
    char const *where = where0;
    char const *lim = where + digs;
    int negative = 0;

    /* Accommodate buggy tar of unknown vintage, which outputs leading
       NUL if the previous field overflows. */
    where += !*where;

    /* Accommodate older tars, which output leading spaces. */
    for (;;)
    {
        if (where == lim)
        {
            if (type && !silent)
                ERROR ((0, 0,
                       /* TRANSLATORS: %s is type of the value (gid_t, uid_t, etc.) */
                       _("Blanks in header where numeric %s value expected"),
                       type));
            return -1;
        }
        if (!ISSPACE ((unsigned char) *where))
            break;
        where++;
    }

    value = 0;
    if (ISODIGIT (*where))
    {
        char const *where1 = where;
        uintmax_t overflow = 0;

```

```

for (;;)
{
    value += *where++ - '0';
    if (where == lim || ! ISODIGIT (*where))
        break;
    overflow |= value ^ (value << LG_8 >> LG_8);
    value <<= LG_8;
}

/* Parse the output of older, unportable tars, which generate
negative values in two's complement octal. If the leading
nonzero digit is 1, we can't recover the original value
reliably; so do this only if the digit is 2 or more. This
catches the common case of 32-bit negative time stamps. */
if ((overflow || maxval < value) && '2' <= *where1 && type)
{
    /* Compute the negative of the input value, assuming two's
    complement. */
    int digit = (*where1 - '0') | 4;
    overflow = 0;
    value = 0;
    where = where1;
    for (;;)
    {
        value += 7 - digit;
        where++;
        if (where == lim || ! ISODIGIT (*where))
            break;
        digit = *where - '0';
        overflow |= value ^ (value << LG_8 >> LG_8);
        value <<= LG_8;
    }
    value++;
    overflow |= !value;

    if (!overflow && value <= minus_minval)
    {
        if (!silent)
            WARN ((0, 0,
                /* TRANSLATORS: Second %s is a type name (gid_t,uid_t,etc.) */
                _("Archive octal value %.*s is out of %s range; assuming two's complement"),
                (int) (where - where1), where1, type));
        negative = 1;
    }
}

if (overflow)
{
    if (type && !silent)
        ERROR ((0, 0,
            /* TRANSLATORS: Second %s is a type name (gid_t,uid_t,etc.) */
            _("Archive octal value %.*s is out of %s range"),

```

```

        (int) (where - where1), where1, type));
    return -1;
}
}
else if (octal_only)
{
    /* Suppress the following extensions. */
}
else if (*where == '-' || *where == '+')
{
    /* Parse base-64 output produced only by tar test versions
       1.13.6 (1999-08-11) through 1.13.11 (1999-08-23).
       Support for this will be withdrawn in future releases. */
    int dig;
    if (!silent)
    {
        static bool warned_once;
        if (! warned_once)
        {
            warned_once = true;
            WARN ((0, 0, _("Archive contains obsolescent base-64 headers")));
        }
    }
    negative = *where++ == '-';
    while (where != lim
           && (dig = base64_map[(unsigned char) *where] < 64)
           {
        if (value << LG_64 >> LG_64 != value)
        {
            char *string = alloca (digs + 1);
            memcpy (string, where0, digs);
            string[digs] = '\0';
            if (type && !silent)
                ERROR ((0, 0,
                       _("Archive signed base-64 string %s is out of %s range"),
                       quote (string), type));
            return -1;
        }
        value = (value << LG_64) | dig;
        where++;
    }
}
else if (*where == '\200' /* positive base-256 */
        || *where == '\377' /* negative base-256 */)
{
    /* Parse base-256 output. A nonnegative number N is
       represented as (256**DIGS)/2 + N; a negative number -N is
       represented as (256**DIGS) - N, i.e. as two's complement.
       The representation guarantees that the leading bit is
       always on, so that we don't confuse this format with the
       others (assuming ASCII bytes of 8 bits or more). */
    int signbit = *where & (1 << (LG_256 - 2));

```

```

uintmax_t topbits = (((uintmax_t) - signbit)
                    << (CHAR_BIT * sizeof (uintmax_t)
                        - LG_256 - (LG_256 - 2)));
value = (*where++ & ((1 << (LG_256 - 2)) - 1)) - signbit;
for (;;)
{
    value = (value << LG_256) + (unsigned char) *where++;
    if (where == lim)
        break;
    if (((value << LG_256 >> LG_256) | topbits) != value)
    {
        if (type && !silent)
            ERROR ((0, 0,
                    _("Archive base-256 value is out of %s range"),
                    type));
        return -1;
    }
}
negative = signbit;
if (negative)
    value = -value;
}

if (where != lim && *where && !ISSPACE ((unsigned char) *where))
{
    if (type)
    {
        char buf[1000]; /* Big enough to represent any header. */
        static struct quoting_options *o;

        if (!o)
        {
            o = clone_quoting_options (0);
            set_quoting_style (o, locale_quoting_style);
        }

        while (where0 != lim && ! lim[-1])
            lim--;
        quotearg_buffer (buf, sizeof buf, where0, lim - where, o);
        if (!silent)
            ERROR ((0, 0,
                    /* TRANSLATORS: Second %s is a type name (gid_t,uid_t,etc.) */
                    _("Archive contains %.*s where numeric %s value expected"),
                    (int) sizeof buf, buf, type));
    }

    return -1;
}

if (value <= (negative ? minus_minval : maxval))
    return negative ? -value : value;

```

```

if (type && !silent)
{
    char minval_buf[UINTMAX_STRSIZE_BOUND + 1];
    char maxval_buf[UINTMAX_STRSIZE_BOUND];
    char value_buf[UINTMAX_STRSIZE_BOUND + 1];
    char *minval_string = STRINGIFY_BIGINT (minus_minval, minval_buf + 1);
    char *value_string = STRINGIFY_BIGINT (value, value_buf + 1);
    if (negative)
        *--value_string = '-';
    if (minus_minval)
        *--minval_string = '-';
    /* TRANSLATORS: Second %s is type name (gid_t,uid_t,etc.) */
    ERROR ((0, 0, _("Archive value %s is out of %s range %s..%s"),
           value_string, type,
           minval_string, STRINGIFY_BIGINT (maxval, maxval_buf)));
}

return -1;
}

gid_t
gid_from_header (const char *p, size_t s)
{
    return from_header (p, s, "gid_t",
                       - (uintmax_t) TYPE_MINIMUM (gid_t),
                       (uintmax_t) TYPE_MAXIMUM (gid_t),
                       false, false);
}

major_t
major_from_header (const char *p, size_t s)
{
    return from_header (p, s, "major_t",
                       - (uintmax_t) TYPE_MINIMUM (major_t),
                       (uintmax_t) TYPE_MAXIMUM (major_t), false, false);
}

minor_t
minor_from_header (const char *p, size_t s)
{
    return from_header (p, s, "minor_t",
                       - (uintmax_t) TYPE_MINIMUM (minor_t),
                       (uintmax_t) TYPE_MAXIMUM (minor_t), false, false);
}

mode_t
mode_from_header (const char *p, size_t s)
{
    /* Do not complain about unrecognized mode bits. */
    unsigned u = from_header (p, s, "mode_t",
                              - (uintmax_t) TYPE_MINIMUM (mode_t),
                              TYPE_MAXIMUM (uintmax_t), false, false);
}

```

```

return ((u & TSUID ? S_ISUID : 0)
        | (u & TSGID ? S_ISGID : 0)
        | (u & TSVTX ? S_ISVTX : 0)
        | (u & TUREAD ? S_IRUSR : 0)
        | (u & TUWRITE ? S_IWUSR : 0)
        | (u & TUEXEC ? S_IXUSR : 0)
        | (u & TGREAD ? S_IRGRP : 0)
        | (u & TGWRITE ? S_IWGRP : 0)
        | (u & TGEXEC ? S_IXGRP : 0)
        | (u & TOREAD ? S_IROTH : 0)
        | (u & TOWRITE ? S_IWOTH : 0)
        | (u & TOEXEC ? S_IXOTH : 0));
}

off_t
off_from_header (const char *p, size_t s)
{
    /* Negative offsets are not allowed in tar files, so invoke
       from_header with minimum value 0, not TYPE_MINIMUM (off_t). */
    return from_header (p, s, "off_t", (uintmax_t) 0,
                       (uintmax_t) TYPE_MAXIMUM (off_t), false, false);
}

size_t
size_from_header (const char *p, size_t s)
{
    return from_header (p, s, "size_t", (uintmax_t) 0,
                       (uintmax_t) TYPE_MAXIMUM (size_t), false, false);
}

time_t
time_from_header (const char *p, size_t s)
{
    return from_header (p, s, "time_t",
                       - (uintmax_t) TYPE_MINIMUM (time_t),
                       (uintmax_t) TYPE_MAXIMUM (time_t), false, false);
}

uid_t
uid_from_header (const char *p, size_t s)
{
    return from_header (p, s, "uid_t",
                       - (uintmax_t) TYPE_MINIMUM (uid_t),
                       (uintmax_t) TYPE_MAXIMUM (uid_t), false, false);
}

uintmax_t
uintmax_from_header (const char *p, size_t s)
{
    return from_header (p, s, "uintmax_t", (uintmax_t) 0,
                       TYPE_MAXIMUM (uintmax_t), false, false);
}

```

```

/* Return a printable representation of T. The result points to
static storage that can be reused in the next call to this
function, to ctime, or to asctime. If FULL_TIME, then output the
time stamp to its full resolution; otherwise, just output it to
1-minute resolution. */
char const *
tartime (struct timespec t, bool full_time)
{
    enum { fraclen = sizeof ".FFFFFFFF" - 1 };
    static char buffer[max (UINTMAX_STRSIZE_BOUND + 1,
                            INT_STRLEN_BOUND (int) + 16)
                    + fraclen];
    struct tm *tm;
    time_t s = t.tv_sec;
    int ns = t.tv_nsec;
    bool negative = s < 0;
    char *p;

    if (negative && ns != 0)
    {
        s++;
        ns = 1000000000 - ns;
    }

    tm = utc_option ? gmtime (&s) : localtime (&s);
    if (tm)
    {
        if (full_time)
        {
            sprintf (buffer, "%04ld-%02d-%02d %02d:%02d:%02d",
                    tm->tm_year + 1900L, tm->tm_mon + 1, tm->tm_mday,
                    tm->tm_hour, tm->tm_min, tm->tm_sec);
            code_ns_fraction (ns, buffer + strlen (buffer));
        }
        else
            sprintf (buffer, "%04ld-%02d-%02d %02d:%02d",
                    tm->tm_year + 1900L, tm->tm_mon + 1, tm->tm_mday,
                    tm->tm_hour, tm->tm_min);
        return buffer;
    }

    /* The time stamp cannot be broken down, most likely because it
is out of range. Convert it as an integer,
right-adjusted in a field with the same width as the usual
4-year ISO time format. */
    p = umaxtostr (negative ? - (uintmax_t) s : s,
                  buffer + sizeof buffer - UINTMAX_STRSIZE_BOUND - fraclen);
    if (negative)
        *--p = '-';
    while ((buffer + sizeof buffer - sizeof "YYYY-MM-DD HH:MM"

```

```

        + (full_time ? sizeof ":SS.FFFFFFFF" - 1 : 0))
    < p)
    *--p = ' ';
if (full_time)
    code_ns_fraction (ns, buffer + sizeof buffer - 1 - fraclen);
return p;
}

```

/* Actually print it.

Plain and fancy file header block logging. Non-verbose just prints the name, e.g. for "tar t" or "tar x". This should just contain file names, so it can be fed back into tar with xargs or the "-T" option. The verbose option can give a bunch of info, one line per file. I doubt anybody tries to parse its format, or if they do, they shouldn't. Unix tar is pretty random here anyway. */

/* FIXME: Note that print_header uses the globals HEAD, HSTAT, and HEAD_STANDARD, which must be set up in advance. Not very clean.. */

/* Width of "user/group size", with initial value chosen heuristically. This grows as needed, though this may cause some stairstepping in the output. Make it too small and the output will almost always look ragged. Make it too large and the output will be spaced out too far. */

```
static int ugswidth = 19;
```

/* Width of printed time stamps. It grows if longer time stamps are found (typically, those with nanosecond resolution). Like USGWIDTH, some stairstepping may occur. */

```
static int datewidth = sizeof "YYYY-MM-DD HH:MM" - 1;
```

```
void
```

```
print_header (struct tar_stat_info *st, off_t block_ordinal)
```

```
{
    char modes[11];
    char const *time_stamp;
    int time_stamp_len;
    char *temp_name;
```

/* These hold formatted ints. */

```
char uform[UINTMAX_STRSIZE_BOUND], gform[UINTMAX_STRSIZE_BOUND];
```

```
char *user, *group;
```

```
char size[2 * UINTMAX_STRSIZE_BOUND];
```

/* holds formatted size or major,minor */

```
char uintbuf[UINTMAX_STRSIZE_BOUND];
```

```
int pad;
```

```
int sizelen;
```

```
if (test_label_option && current_header->header.typeflag != GNUTYPE_VOLHDR)
    return;
```

```

if (show_transformed_names_option)
    temp_name = st->file_name ? st->file_name : st->orig_file_name;
else
    temp_name = st->orig_file_name ? st->orig_file_name : st->file_name;

if (block_number_option)
{
    char buf[UINTMAX_STRSIZE_BOUND];
    if (block_ordinal < 0)
        block_ordinal = current_block_ordinal ();
    block_ordinal -= recent_long_name_blocks;
    block_ordinal -= recent_long_link_blocks;
    fprintf (stdlis, _("block %s: "),
             STRINGIFY_BIGINT (block_ordinal, buf));
}

if (verbose_option <= 1)
{
    /* Just the fax, mam. */
    fprintf (stdlis, "%s\n", quotearg (temp_name));
}
else
{
    /* File type and modes. */

    modes[0] = '?';
    switch (current_header->header.typeflag)
    {
        case GNUTYPE_VOLHDR:
            modes[0] = 'V';
            break;

        case GNUTYPE_MULTIVOL:
            modes[0] = 'M';
            break;

        case GNUTYPE_LONGNAME:
        case GNUTYPE_LONGLINK:
            modes[0] = 'L';
            ERROR ((0, 0, _("Unexpected long name header")));
            break;

        case GNUTYPE_SPARSE:
        case REGTYPE:
        case AREGTYPE:
            modes[0] = '-';
            if (temp_name[strlen (temp_name) - 1] == '/')
                modes[0] = 'd';
            break;
        case LNKTYPE:
            modes[0] = 'h';

```

```

    break;
case GNUTYPE_DUMPDIR:
    modes[0] = 'd';
    break;
case DIRTYE:
    modes[0] = 'd';
    break;
case SYMTYPE:
    modes[0] = 'l';
    break;
case BLKTYPE:
    modes[0] = 'b';
    break;
case CHRTYPE:
    modes[0] = 'c';
    break;
case FIFOTYPE:
    modes[0] = 'p';
    break;
case CONTTYPE:
    modes[0] = 'C';
    break;
}

```

```
pax_decode_mode (st->stat.st_mode, modes + 1);
```

```
/* Time stamp. */
```

```

time_stamp = tartime (st->mtime, false);
time_stamp_len = strlen (time_stamp);
if (datewidth < time_stamp_len)
    datewidth = time_stamp_len;

```

```
/* User and group names. */
```

```

if (st->uname
    && st->uname[0]
    && current_format != V7_FORMAT
    && !numeric_owner_option)
    user = st->uname;
else
    {
    /* Try parsing it as an unsigned integer first, and as a
       uid_t if that fails. This method can list positive user
       ids that are too large to fit in a uid_t. */
    uintmax_t u = from_header (current_header->header.uid,
                              sizeof current_header->header.uid, 0,
                              (uintmax_t) 0,
                              (uintmax_t) TYPE_MAXIMUM (uintmax_t),
                              false, false);

    if (u != -1)
        user = STRINGIFY_BIGINT (u, uform);
    }

```

```

else
{
    sprintf (uform, "%ld",
            (long) UID_FROM_HEADER (current_header->header.uid));
    user = uform;
}
}

if (st->gname
    && st->gname[0]
    && current_format != V7_FORMAT
    && !numeric_owner_option)
    group = st->gname;
else
{
    /* Try parsing it as an unsigned integer first, and as a
       gid_t if that fails. This method can list positive group
       ids that are too large to fit in a gid_t. */
    uintmax_t g = from_header (current_header->header.gid,
                              sizeof current_header->header.gid, 0,
                              (uintmax_t) 0,
                              (uintmax_t) TYPE_MAXIMUM (uintmax_t),
                              false, false);

    if (g != -1)
        group = STRINGIFY_BIGINT (g, gform);
    else
    {
        sprintf (gform, "%ld",
                (long) GID_FROM_HEADER (current_header->header.gid));
        group = gform;
    }
}

/* Format the file size or major/minor device numbers. */

switch (current_header->header.typeflag)
{
    case CHRTYPE:
    case BLKTYPE:
        strcpy (size,
                STRINGIFY_BIGINT (major (st->stat.st_rdev), uintbuf));
        strcat (size, ",");
        strcat (size,
                STRINGIFY_BIGINT (minor (st->stat.st_rdev), uintbuf));
        break;

    default:
        /* st->stat.st_size keeps stored file size */
        strcpy (size, STRINGIFY_BIGINT (st->stat.st_size, uintbuf));
        break;
}

```

```

/* Figure out padding and print the whole line. */

sizelen = strlen (size);
pad = strlen (user) + 1 + strlen (group) + 1 + sizelen;
if (pad > ugswidth)
    ugswidth = pad;

fprintf (stdlis, "%s %s/%s %*s %-*s",
         modes, user, group, ugswidth - pad + sizelen, size,
         datewidth, time_stamp);

fprintf (stdlis, " %s", quotearg (temp_name));

switch (current_header->header.typeflag)
{
case SYMTYPE:
    fprintf (stdlis, "-> %s\n", quotearg (st->link_name));
    break;

case LNKTYPE:
    fprintf (stdlis, _(" link to %s\n"), quotearg (st->link_name));
    break;

default:
    {
    char type_string[2];
    type_string[0] = current_header->header.typeflag;
    type_string[1] = '\0';
    fprintf (stdlis, _(" unknown file type %s\n"),
            quote (type_string));
    }
    break;

case AREGTYPE:
case REGTYPE:
case GNUTYPE_SPARSE:
case CHRTYPE:
case BLKTYPE:
case DIRTYTYPE:
case FIFOTYPE:
case CONTTYPE:
case GNUTYPE_DUMPDIR:
    putc ('\n', stdlis);
    break;

case GNUTYPE_LONGLINK:
    fprintf (stdlis, _("--Long Link--\n"));
    break;

case GNUTYPE_LONGNAME:
    fprintf (stdlis, _("--Long Name--\n"));
    break;
}

```

```

case GNUTYPE_VOLHDR:
    fprintf (stdlis, _("--Volume Header--\n"));
    break;

case GNUTYPE_MULTIVOL:
    strcpy (size,
            STRINGIFY_BIGINT
            (UINTMAX_FROM_HEADER (current_header->oldgnu_header.offset),
            uintbuf));
    fprintf (stdlis, _("--Continued at byte %s--\n"), size);
    break;
    }
}
fflush (stdlis);
}

```

/* Print a similar line when we make a directory automatically. */

```

void
print_for_mkdir (char *dirname, int length, mode_t mode)
{
    char modes[11];

    if (verbose_option > 1)
    {
        /* File type and modes. */

        modes[0] = 'd';
        pax_decode_mode (mode, modes + 1);

        if (block_number_option)
        {
            char buf[UINTMAX_STRSIZE_BOUND];
            fprintf (stdlis, _("block %s: "),
                    STRINGIFY_BIGINT (current_block_ordinal (), buf));
        }

        fprintf (stdlis, "%s %*s %.*s\n", modes, ugswidth + 1 + datewidth,
                _("Creating directory:"), length, quotearg (dirname));
    }
}

```

/* Skip over SIZE bytes of data in blocks in the archive. */

```

void
skip_file (off_t size)
{
    union block *x;

    /* FIXME: Make sure mv_begin is always called before it */

    if (seekable_archive)
    {

```

```

    off_t nblk = seek_archive (size);
    if (nblk >= 0)
        size -= nblk * BLOCKSIZE;
    else
        seekable_archive = false;
}

mv_size_left (size);

while (size > 0)
{
    x = find_next_block ();
    if (! x)
        FATAL_ERROR ((0, 0, _("Unexpected EOF in archive")));

    set_next_block_after (x);
    size -= BLOCKSIZE;
    mv_size_left (size);
}

/* Skip the current member in the archive.
   NOTE: Current header must be decoded before calling this function. */
void
skip_member (void)
{
    if (!current_stat_info.skipped)
    {
        char save_typeflag = current_header->header.typeflag;
        set_next_block_after (current_header);

        mv_begin (&current_stat_info);

        if (current_stat_info.is_sparse)
            sparse_skip_file (&current_stat_info);
        else if (save_typeflag != DIRTYPE)
            skip_file (current_stat_info.stat.st_size);

        mv_end ();
    }
}

```