

new/usr/src/cmd/tar/tar.c

```
*****
238690 Sun Aug 16 13:00:01 2015
new/usr/src/cmd/tar/tar.c
6128 tar should check prefix field when detecting EOT
6129 tar debug output should be available in all builds
Reviewed by: Robert Mustacchi <rm@joyent.com>
*****
1 /*
2  * CDDL HEADER START
3  *
4  * The contents of this file are subject to the terms of the
5  * Common Development and Distribution License (the "License").
6  * You may not use this file except in compliance with the License.
7  *
8  * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
9  * or http://www.opensolaris.org/os/licensing.
10 * See the License for the specific language governing permissions
11 * and limitations under the License.
12 *
13 * When distributing Covered Code, include this CDDL HEADER in each
14 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
15 * If applicable, add the following below this CDDL HEADER, with the
16 * fields enclosed by brackets "[]" replaced with your own identifying
17 * information: Portions Copyright [yyyy] [name of copyright owner]
18 *
19 * CDDL HEADER END
20 */
21 /*
22 * Copyright (c) 1988, 2010, Oracle and/or its affiliates. All rights reserved.
23 * Copyright 2012 Milan Jurik. All rights reserved.
24 * Copyright 2015 Joyent, Inc.
24 * Copyright (c) 2013, Joyent, Inc. All rights reserved.
25 */

27 /* Copyright (c) 1983, 1984, 1985, 1986, 1987, 1988, 1989 AT&T      */
28 /* All Rights Reserved */
30 /* Copyright (c) 1987, 1988 Microsoft Corporation */
31 /* All Rights Reserved */

33 /*
34 * Portions of this source code were derived from Berkeley 4.3 BSD
35 * under license from the Regents of the University of California.
36 */

38 #include <unistd.h>
39 #include <sys/types.h>
40 #include <sys/param.h>
41 #include <sys/stat.h>
42 #include <sys/mkdev.h>
43 #include <sys/wait.h>
44 #include <dirent.h>
45 #include <errno.h>
46 #include <stdio.h>
47 #include <signal.h>
48 #include <ctype.h>
49 #include <locale.h>
50 #include <nl_types.h>
51 #include <langinfo.h>
52 #include <pwd.h>
53 #include <grp.h>
54 #include <fcntl.h>
55 #include <string.h>
56 #include <malloc.h>
57 #include <time.h>
58 #include <utime.h>
```

1

new/usr/src/cmd/tar/tar.c

```
59 #include <stdlib.h>
60 #include <stdarg.h>
61 #include <widec.h>
62 #include <sys/mtio.h>
63 #include <sys/acl.h>
64 #include <strings.h>
65 #include <deflt.h>
66 #include <limits.h>
67 #include <iconv.h>
68 #include <assert.h>
69 #include <libgen.h>
70 #include <libintl.h>
71 #include <aclutils.h>
72 #include <libnvpair.h>
73 #include <archives.h>

75 #if defined(__SunOS_5_6) || defined(__SunOS_5_7)
76 extern int defcntl();
77#endif
78 #if defined(_PC_SATTR_ENABLED)
79 #include <attr.h>
80 #include <libcmdutils.h>
81#endif

83 /* Trusted Extensions */
84 #include <zone.h>
85 #include <tsol/label.h>
86 #include <sys/tsol/label_macro.h>

88 #include "getresponse.h"
89 /*
90 * Source compatibility
91 */

93 /*
94 * These constants come from archives.h and sys/fcntl.h
95 * and were introduced by the extended attributes project
96 * in Solaris 9.
97 */
98 #if !defined(O_XATTR)
99 #define AT_SYMLINK_NOFOLLOW 0x1000
100 #define AT_REMOVEDIR 0x1
101 #define AT_FDCWD 0xffffd19553
102 #define _XATTR_HDRTYPE 'E'
103 static int attropen();
104 static int fstatat();
105 static int renameat();
106 static int unlinkat();
107 static int openat();
108 static int fchownat();
109 static int futimesat();
110#endif

112 /*
113 * Compiling with -D_XPG4_2 gets this but produces other problems, so
114 * instead of including sys/time.h and compiling with -D_XPG4_2, I'm
115 * explicitly doing the declaration here.
116 */
117 int utimes(const char *path, const struct timeval timeval_ptr[]);

119 #ifndef MINSIZE
120 #define MINSIZE 250
121#endif
122#define DEF_FILE "/etc/default/tar"

124#define min(a, b) ((a) < (b) ? (a) : (b))
```

2

```

125 #define max(a, b) ((a) > (b) ? (a) : (b))

127 /* -DDEBUG ONLY for debugging */
128 #ifdef DEBUG
129 #undef DEBUG
130 #define DEBUG(a, b, c)\n    (void) fprintf(stderr, "DEBUG - "), (void) fprintf(stderr, a, b, c)
131 #endif

127 #define TBLOCK 512      /* tape block size--should be universal */

129 #define BSIZE
130 #define SYS_BLOCK BSIZE /* from sys/param.h: secondary block size */
131 #else /* BSIZE */
132 #define SYS_BLOCK 512   /* default if no BSIZE in param.h */
133 #endif /* BSIZE */

135 #define NBLOCK 20
136 #define NAMSIZ 100
137 #define PRESIZ 155
138 #define MAXNAM 256
139 #define MODEMASK 07777777      /* file creation mode mask */
140 #define POSIXMODES 07777      /* mask for POSIX mode bits */
141 #define MAXEXT 9           /* reasonable max # extents for a file */
142 #define EXTMIN 50          /* min blks left on floppy to split a file */

144 /* max value dblockdbuf.efsize can store */
145 #define TAR_EFSIZE_MAX 0777777777

147 /*
148 * Symbols which specify the values at which the use of the 'E' function
149 * modifier is required to properly store a file.
150 *
151 *     TAR_OFFSET_MAX - the largest file size we can archive
152 *     OCTAL7CHAR - the limit for ustar gid, uid, dev
153 */

155 #ifdef XHDR_DEBUG
156 /* tiny values which force the creation of extended header entries */
157 #define TAR_OFFSET_MAX 9
158 #define OCTAL7CHAR 2
159 #else
160 /* normal values */
161 #define TAR_OFFSET_MAX 077777777777ULL
162 #define OCTAL7CHAR 07777777
163 #endif

165 #define TBLOCKS(bytes) (((bytes) + TBLOCK - 1) / TBLOCK)
166 #define K(tblocks) ((tblocks+1)/2) /* tblocks to Kbytes for printing */

168 #define MAXLEV (PATH_MAX / 2)
169 #define LEV0 1
170 #define SYMLINK_LEV0 0

172 #define TRUE 1
173 #define FALSE 0

175 #define XATTR_FILE 1
176 #define NORMAL_FILE 0

178 #define PUT_AS_LINK 1
179 #define PUT_NOTAS_LINK 0

181 #ifndef VIEW_READONLY
182 #define VIEW_READONLY "SUNWattr_rw"
183 #endif

```

```

185 #ifndef VIEW_READWRITE
186 #define VIEW_READWRITE "SUNWattr_rw"
187 #endif

189 #if _FILE_OFFSET_BITS == 64
190 #define FMT_off_t "lld"
191 #define FMT_off_t_o "lo"
192 #define FMT_blkcnt_t "lld"
193 #else
194 #define FMT_off_t "ld"
195 #define FMT_off_t_o "lo"
196 #define FMT_blkcnt_t "ld"
197 #endif

199 /* ACL support */

201 static struct sec_attr {
202     char attr_type;
203     char attr_len[7];
204     char attr_info[1];
205 } *attr;
unchanged_portion_omitted
417 static file_list_t *exclude_tbl[TABLE_SIZE],
418                 *include_tbl[TABLE_SIZE];

420 static int append_secattr(char **, int *, int, char *, char);
421 static void write_ancillary(union hblock *, char *, int, char);

423 static void add_file_to_table(file_list_t *table[], char *str);
424 static void assert_string(char *s, char *msg);
425 static int istape(int fd, int type);
426 static void backtape(void);
427 static void build_table(file_list_t *table[], char *file);
428 static int check_prefix(char **namep, char **dirp, char **compp);
429 static void closevol(void);
430 static void copy(void *dst, void *src);
431 static int convtoreg(off_t);
432 static void delete_target(int fd, char *comp, char *namep);
433 static void doBirTimes(char *name, timestamp_t modTime);
434 static void done(int n);
435 static void dorep(char *argv[]);
436 static void dotable(char *argv[]);
437 static void dosubtract(char *argv[]);
438 static int tar_chdir(const char *path);
439 static int is_directory(char *name);
440 static int has_dot_dot(char *name);
441 static int is_absolute(char *name);
442 static char *make_relative_name(char *name, char **stripped_prefix);
443 static void fatal(char *format, ...);
444 static void v perror(int exit_status, char *fmt, ...);
445 static void flushtape(void);
446 static void getdir(void);
447 static void *getmem(size_t);
448 static void longt(struct stat *st, char aclchar);
449 static void load_info_from_xtarhdr(u_longlong_t flag, struct xtar_hdr *xhdrp);
450 static int makeDir(char *name);
451 static void mterr(char *operation, int i, int exitcode);
452 static void newvol(void);
453 static void passtape(void);
454 static void putempty(blkcnt_t n);
455 static int putfile(char *longname, char *shortname, char *parent,
456                   attr_data_t *attrinfo, int filetype, int lev, int symlink_lev);
457 static void readtape(char *buffer);
458 static void seekdisk(blkcnt_t blocks);

```

```

459 static void setPathTimes(int dirfd, char *path, timestruc_t modTime);
460 static void setbytes_to_skip(struct stat *st, int err);
461 static void splitfile(char *longname, int ifd, char *name,
462     char *prefix, int filetype);
463 static void tomodes(struct stat *sp);
464 static void usage(void);
465 static int xblocks(int issysattr, off_t bytes, int ofile);
466 static int xsfile(int issysattr, int ofd);
467 static void resugname(int dirfd, char *name, int symflag);
468 static int bcheck(char *bstr);
469 static int checkdir(char *name);
470 static int checksum(union hblock *dblockp);
471 #ifdef EUC
472 static int checksum_signed(union hblock *dblockp);
473 #endif /* EUC */
474 static int checkupdate(char *arg);
475 static int checkw(char c, char *name);
476 static int cmp(char *b, char *s, int n);
477 static int defset(char *arch);
478 static boolean_t endtape(void);
479 static int is_in_table(file_list_t *table[], char *str);
480 static int notsame(void);
481 static int is_prefix(char *s1, char *s2);
482 static int response(void);
483 static int build_dblock(const char *, const char *, const char,
484     const int filetype, const struct stat *, const dev_t, const char *);
485 static unsigned int hash(char *str);

487 static blkcnt_t kcheck(char *kstr);
488 static off_t bsrch(char *s, int n, off_t l, off_t h);
489 static void onquit(int sig);
490 static void onhup(int sig);
491 static void uid_t getuidbyname(char *);
492 static uid_t getgidbyname(char *);
493 static char *getname(gid_t);
494 static char *getgroup(gid_t);
495 static int checkf(char *name, int mode, int howmuch);
496 static int writetbuf(char *buffer, int n);
498 static int wantit(char *argv[], char **namep, char **dirp, char **comp,
499     attr_data_t **attrinfo);
500 static void append_ext_attr(char *shortname, char **secinfo, int *len);
501 static int get_xdata(void);
502 static void gen_num(const char *keyword, const u_longlong_t number);
503 static void gen_date(const char *keyword, const timestruc_t time_value);
504 static void gen_string(const char *keyword, const char *value);
505 static void get_xtime(char *value, timestruc_t *xtime);
506 static int chk_path_build(char *name, char *longname, char *linkname,
507     char *prefix, char type, int filetype);
508 static int gen_utf8_names(const char *filename);
509 static int utf8_local(char *option, char **Xhdr_ptrptr, char *target,
510     const char *src, int max_val);
511 static int local_utf8(char **Xhdr_ptrptr, char *target, const char *src,
512     iconv_t iconv_cd, int xhdrflg, int max_val);
513 static int c_utf8(char *target, const char *source);
514 static int getstat(int dirfd, char *longname, char *shortname,
515     char *attrparent);
516 static void xattrs_put(char *, char *, char *, char *);
517 static void prepare_xattr(char **, char *, char *,
518     char, struct linkbuf *, int *);
519 static int put_link(char *name, char *longname, char *component,
520     char *longattrname, char *prefix, int filetype, char typeflag);
521 static int put_extra_attributes(char *longname, char *shortname,
522     char *longattrname, char *prefix, int filetype, char typeflag);
523 static int put_xattr_hdr(char *longname, char *shortname, char *longattrname,

```

```

524     char *prefix, int typeflag, int filetype, struct linkbuf *lp);
525 static int read_xattr_hdr(attr_data_t **attrinfo);
526 /* Trusted Extensions */
528 #define AUTO_ZONE           "/zone"
529 static void extract_attr(char **file_ptr, struct sec_attr *);
530 static int check_ext_attr(char *filename);
532 static void rebuild_comp_path(char *str, char **namep);
533 static int rebuild_lk_comp_path(char *str, char **namep);
535 static void get_parent(char *path, char *dir);
536 static char *get_component(char *path);
537 static int retry_open_attr(int pdirfd, int cwd, char *dirp, char *pattr,
538     char *name, int oflag, mode_t mode);
539 static char *skipslashes(char *string, char *start);
540 static void chop_endslashes(char *path);
541 static pid_t compress_file(void);
542 static void compress_back(void);
543 static void decompress_file(void);
544 static pid_t uncompress_file(void);
545 static void *compress_malloc(size_t);
546 static void check_compression(void);
547 static char *bz_suffix(void);
548 static char *gz_suffix(void);
549 static char *xz_suffix(void);
550 static char *add_suffix();
551 static void wait_pid(pid_t);
552 static void verify_compress_opt(const char *t);
553 static void detect_compress(void);
554 static void dlog(const char *, ...);
555 static boolean_t should_enable_debug(void);

557 static struct stat stbuf;
559 static char *myname;
560 static char *xtract_chdir = NULL;
561 static int checkflag = 0;
562 static int Xflag, Fflag, iflag, hflag, Bflag, Iflag;
563 static int rflag, xflag, vflag, tflag, mt, cflag, mflag, pflag;
564 static int uflag;
565 static int errflag;
566 static int oflag;
567 static int bflag, Aflag;
568 static int Pflag;          /* POSIX conformant archive */
569 static int Eflag;          /* Allow files greater than 8GB */
570 static int atflag;         /* traverse extended attributes */
571 static int saflag;         /* traverse extended sys attributes */
572 static int Dflag;          /* Data change flag */
573 static int jflag;          /* flag to use 'bzip2' */
574 static int zflag;          /* flag to use 'gzip' */
575 static int Zflag;          /* flag to use 'compress' */
576 static int Jflag;          /* flag to use 'xz' */
577 static int aflag;          /* flag to use autocompression */

579 /* Trusted Extensions */
580 static int Tflag;          /* Trusted Extensions attr flags */
581 static int dir_flag;        /* for attribute extract */
582 static int mld_flag;        /* for attribute extract */
583 static char *orig_namep;    /* original namep - unadorned */
584 static int rpath_flag;      /* MLD real path is rebuilt */
585 static char real_path[MAXPATHLEN]; /* MLD real path */
586 static int lk_rpath_flag;   /* linked to real path is rebuilt */
587 static char lk_real_path[MAXPATHLEN]; /* linked real path */
588 static bslabel_t bs_label;  /* for attribute extract */
589 static bslabel_t admin_low;
```

```

590 static bslabel_t admin_high;
591 static int ignored_aprivils = 0;
592 static int ignored_fprivs = 0;
593 static int ignored_fatptrs = 0;

595 static int term, chksum, wflag,
596 first = TRUE, defaults_used = FALSE, linkerrok;
597 static blkcnt_t recno;
598 static int freemem = 1;
599 static int nblock = NBLOCK;
600 static int Errflg = 0;
601 static int exitflag = 0;

603 static dev_t mt_dev; /* device containing output file */
604 static ino_t mt_ino; /* inode number of output file */
605 static int mt_devtype; /* dev type of archive, from stat structure */

607 static int update = 1; /* for 'open' call */

609 static off_t low;
610 static off_t high;

612 static FILE *tfile;
613 static FILE *vfile = stdout;
614 static char *tmpdir;
615 static char *tmp_suffix = "/tarXXXXXX";
616 static char *name;
617 static char archive[] = "archive0=";
618 static char *Xfile;
619 static char *usefile;
620 static char tfname[1024];

622 static int mulvol; /* multi-volume option selected */
623 static blkcnt_t blocklim; /* number of blocks to accept per volume */
624 static blkcnt_t tapepos; /* current block number to be written */
625 static int NotTape; /* true if tape is a disk */
626 static int dumping; /* true if writing a tape or other archive */
627 static int extno; /* number of extent: starts at 1 */
628 static int extotal; /* total extents in this file */
629 static off_t extsize; /* size of current extent during extraction */
630 static ushort_t Oumask = 0; /* old umask value */
631 static boolean_t is_posix; /* true if archive is POSIX-conformant */
632 static const char *magic_type = "ustar";
633 static size_t xrec_size = 8 * PATH_MAX; /* extended rec initial size */
634 static char *xrec_ptr;
635 static off_t xrec_offset = 0;
636 static int Xhdrflag;
637 static int charset_type = 0;

639 static u_longlong_t xhdr_flg; /* Bits set determine which items */
640 /* need to be in extended header. */
641 static pid_t comp_pid = 0;

643 static boolean_t debug_output = B_FALSE;

645 #define _X_DEVMAJOR 0x1
646 #define _X_DEVMINOR 0x2
647 #define _X_GID 0x4
648 #define _X_GNAME 0x8
649 #define _X_LINKPATH 0x10
650 #define _X_PATH 0x20
651 #define _X_SIZE 0x40
652 #define _X_UID 0x80
653 #define _X_UNAME 0x100
654 #define _X_ATIME 0x200

```

```

655 #define _X_CTIME 0x400
656 #define _X_MTIME 0x800
657 #define _X_XHDR 0x1000 /* Bit flag that determines whether 'X' */
658 /* typeflag was followed by 'A' or non 'A' */
659 /* typeflag. */
660 #define _X_LAST 0x40000000

662 #define PID_MAX_DIGITS (10 * sizeof(pid_t) / 4)
663 #define TIME_MAX_DIGITS (10 * sizeof(time_t) / 4)
664 #define LONG_MAX_DIGITS (10 * sizeof(long) / 4)
665 #define ULLONGLONG_MAX_DIGITS (10 * sizeof(u_llonglong_t) / 4)
666 /*
667 * UTF_8 encoding requires more space than the current codeset equivalent.
668 * Currently a factor of 2-3 would suffice, but it is possible for a factor
669 * of 6 to be needed in the future, so for safety, we use that here.
670 */
671 #define UTF_8_FACTOR 6

673 static u_llonglong_t xhdr_count = 0;
674 static char xhdr_dirname[PRESIZ + 1];
675 static char pidchars[PID_MAX_DIGITS + 1];
676 static char *tchar = ""; /* null linkpath */

678 static char local_path[UTF_8_FACTOR * PATH_MAX + 1];
679 static char local_linkpath[UTF_8_FACTOR * PATH_MAX + 1];
680 static char local_gname[UTF_8_FACTOR * _POSIX_NAME_MAX + 1];
681 static char local_uname[UTF_8_FACTOR * _POSIX_NAME_MAX + 1];

683 /*
684 * The following mechanism is provided to allow us to debug tar in complicated
685 * situations, like when it is part of a pipe. The idea is that you compile
686 * with -DWAITAROUND defined, and then add the 'D' function modifier to the
687 * target tar invocation, eg. "tar cdf tarfile file". If stderr is available,
688 * it will tell you to which pid to attach the debugger; otherwise, use ps to
689 * find it. Attach to the process from the debugger, and, *PRESTO*, you are
690 * there!
691 *
692 * Simply assign "waitaround = 0" once you attach to the process, and then
693 * proceed from there as usual.
694 */

696 #ifdef WAITAROUND
697 int waitaround = 0; /* wait for rendezvous with the debugger */
698#endif

700 #define BZIP "/usr/bin/bzip2"
701 #define GZIP "/usr/bin/gzip"
702 #define COMPRESS "/usr/bin/compress"
703 #define XZ "/usr/bin/xz"
704 #define BZCAT "/usr/bin/bzcat"
705 #define GZCAT "/usr/bin/gzcat"
706 #define ZCAT "/usr/bin/zcat"
707 #define XZCAT "/usr/bin/xzcat"
708 #define GSUF 8 /* number of valid 'gzip' suffixes */
709 #define BSUF 4 /* number of valid 'bzip2' suffixes */
710 #define XSUF 1 /* number of valid 'xz' suffixes */

712 static char *compress_opt; /* compression type */

714 static char *gsuffix[] = { ".gz", "-gz", ".z", "-z", "_z", ".Z",
715 ".tgz", ".taz" };
716 static char *bsuffix[] = { ".bz2", ".bz", ".tbz2", ".tbz" };
717 static char *xsuffix[] = { ".xz" };
718 static char *suffix;

```

```

721 int
722 main(int argc, char *argv[])
723 {
724     char          *cp;
725     char          *tmpdirp;
726     pid_t         thispid;
727
728     (void) setlocale(LC_ALL, "");
729 #if !defined(TEXT_DOMAIN) /* Should be defined by cc -D */
730 #define TEXT_DOMAIN "SYS_TEST" /* Use this only if it weren't */
731#endif
732     (void) textdomain(TEXT_DOMAIN);
733     if (argc < 2)
734         usage();
735
736     debug_output = should_enable_debug();
737
738     tfile = NULL;
739     if ((myname = strdup(argv[0])) == NULL) {
740         (void) fprintf(stderr, gettext(
741             "tar: cannot allocate program name\n"));
742         exit(1);
743     }
744
745     if (init_yes() < 0) {
746         (void) fprintf(stderr, gettext(ERR_MSG_INIT_YES),
747                     strerror(errno));
748         exit(2);
749     }
750
751     /*
752      * For XPG4 compatibility, we must be able to accept the "--"
753      * argument normally recognized by getopt; it is used to delimit
754      * the end opt the options section, and so can only appear in
755      * the position of the first argument. We simply skip it.
756     */
757
758     if (strcmp(argv[1], "--") == 0) {
759         argv++;
760         argc--;
761         if (argc < 3)
762             usage();
763     }
764
765     argv[argc] = NULL;
766     argv++;
767
768     /*
769      * Set up default values.
770      * Search the operand string looking for the first digit or an 'f'.
771      * If you find a digit, use the 'archive#' entry in DEF_FILE.
772      * If 'f' is given, bypass looking in DEF_FILE altogether.
773      * If no digit or 'f' is given, still look in DEF_FILE but use '0'.
774     */
775     if ((usefile = getenv("TAPE")) == (char *)NULL) {
776         for (cp = *argv; *cp; ++cp)
777             if (isdigit(*cp) || *cp == 'f')
778                 break;
779         if (*cp != 'f') {
780             archive[7] = (*cp)? *cp: '0';
781             if (!defaults_used = defset(archive)) {
782                 usefile = NULL;
783                 nblock = 1;
784                 blocklim = 0;
785                 NotTape = 0;
786             }
787     }

```

```

787             }
788         }
789         for (cp = *argv++; *cp; cp++)
790             switch (*cp) {
791 #ifdef WAITAROUND
792             case 'D':
793                 /* rendezvous with the debugger */
794                 waitaround = 1;
795                 break;
796 #endif
797             case 'f':
798                 assert_string(*argv, gettext(
799                     "tar: tarfile must be specified with 'f' "
800                     "function modifier\n"));
801                 usefile = *argv++;
802                 break;
803             case 'F':
804                 Fflag++;
805                 break;
806             case 'c':
807                 cflag++;
808                 rflag++;
809                 update = 1;
810                 break;
811 #if defined(O_XATTR)
812             case '@':
813                 atflag++;
814                 break;
815 #endif /* O_XATTR */
816 #if defined(_PC_SATTR_ENABLED)
817             case '/':
818                 saflag++;
819                 break;
820 #endif /* _PC_SATTR_ENABLED */
821             case 'u':
822                 uflag++; /* moved code after signals caught */
823                 rflag++;
824                 update = 2;
825                 break;
826             case 'r':
827                 rflag++;
828                 update = 2;
829                 break;
830             case 'v':
831                 vflag++;
832                 break;
833             case 'w':
834                 wflag++;
835                 break;
836             case 'x':
837                 xflag++;
838                 break;
839             case 'X':
840                 assert_string(*argv, gettext(
841                     "tar: exclude file must be specified with 'X' "
842                     "function modifier\n"));
843                 Xflag = 1;
844                 Xfile = *argv++;
845                 build_table(exclude_tbl, Xfile);
846                 break;
847             case 't':
848                 tflag++;
849                 break;
850             case 'm':
851                 mflag++;
852         }

```

```

853         break;
854     case 'p':
855         pflag++;
856         break;
857     case 'D':
858         Dflag++;
859         break;
860     case '-':
861         /* ignore this silently */
862         break;
863     case '0': /* numeric entries used only for defaults */
864     case '1':
865     case '2':
866     case '3':
867     case '4':
868     case '5':
869     case '6':
870     case '7':
871         break;
872     case 'b':
873         assert_string(*argv, gettext(
874             "tar: blocking factor must be specified "
875             "with 'b' function modifier\n"));
876         bflag++;
877         nblock = bcheck(*argv++);
878         break;
879     case 'n': /* not a magtape (instead of 'k') */
880         NotTape++; /* assume non-magtape */
881         break;
882     case 'l':
883         linkerrok++;
884         break;
885     case 'e':
886         errflag++;
887     case 'o':
888         oflag++;
889         break;
890     case 'h':
891         hflag++;
892         break;
893     case 'i':
894         iflag++;
895         break;
896     case 'B':
897         Bflag++;
898         break;
899     case 'P':
900         Pflag++;
901         break;
902     case 'E':
903         Eflag++;
904         Pflag++; /* Only POSIX archive made */
905         break;
906     case 'T':
907         Tflag++; /* Handle Trusted Extensions attrs */
908         pflag++; /* also set flag for ACL */
909         break;
910     case 'j': /* compression "bzip2" */
911         jflag = 1;
912         break;
913     case 'z': /* compression "gzip" */
914         zflag = 1;
915         break;
916     case 'Z': /* compression "compress" */
917         Zflag = 1;
918         break;

```

```

919         case 'J': /* compression "xz" */
920             Jflag = 1;
921             break;
922         case 'a':
923             aflag = 1; /* autocompression */
924             break;
925         default:
926             (void) fprintf(stderr, gettext(
927                 "tar: %c: unknown function modifier\n"), *cp);
928             usage();
929         }
930
931     if (!rflag && !xflag && !tflag)
932         usage();
933     if ((rflag && xflag) || (xflag && tflag) || (rflag && tflag)) {
934         (void) fprintf(stderr, gettext(
935             "tar: specify only one of [ctxru].\n"));
936         usage();
937     }
938     if (cflag) {
939         if ((jflag + zflag + Zflag + Jflag + aflag) > 1) {
940             (void) fprintf(stderr, gettext(
941                 "tar: specify only one of [ajJZZ] to "
942                 "create a compressed file.\n"));
943             usage();
944         }
945     }
946     /* Trusted Extensions attribute handling */
947     if (Tflag && ((getzoneid() != GLOBAL_ZONEID) ||
948         !is_system_labeled())) {
949         (void) fprintf(stderr, gettext(
950             "tar: the 'T' option is only available with "
951             "'Trusted Extensions' and must be run from "
952             "the global zone.\n"));
953         usage();
954     }
955     if (cflag && *argv == NULL)
956         fatal(gettext("Missing filenames"));
957     if (usefile == NULL)
958         fatal(gettext("device argument required"));
959
960     /* alloc a buffer of the right size */
961     if ((tbuf = (union hblock *)
962         calloc(sizeof(union hblock) * nblock, sizeof(char))) ==
963             (union hblock *)NULL) {
964         (void) fprintf(stderr, gettext(
965             "tar: cannot allocate physio buffer\n"));
966         exit(1);
967     }
968
969     if ((xrec_ptr = malloc(xrec_size)) == NULL) {
970         (void) fprintf(stderr, gettext(
971             "tar: cannot allocate extended header buffer\n"));
972         exit(1);
973     }
974
975 #ifdef WAITAROUND
976     if (waitaround) {
977         (void) fprintf(stderr, gettext("Rendezvous with tar on pid"
978             " %d\n"), getpid());
979
980         while (waitaround) {
981             (void) sleep(10);
982         }
983     }
984 #endif

```

```

986     thispid = getpid();
987     (void) sprintf(pidchars, "%ld", thispid);
988     thispid = strlen(pidchars);

990     if ((tmpdirp = getenv("TMPDIR")) == (char *)NULL)
991         (void) strcpy(xhdr_dirname, "/tmp");
992     else {
993         /*
994          * Make sure that dir is no longer than what can
995          * fit in the prefix part of the header.
996          */
997         if (strlen(tmpdirp) > (size_t)(PRESIZ - thispid - 12)) {
998             (void) strcpy(xhdr_dirname, "/tmp");
999             if ((vflag > 0) && (Eflag > 0))
1000                 (void) fprintf(stderr, gettext(
1001                         "Ignoring TMPDIR\n"));
1002         } else
1003             (void) strcpy(xhdr_dirname, tmpdirp);
1004     }
1005     strcat(xhdr_dirname, "/PaxHeaders.");
1006     (void) strcat(xhdr_dirname, pidchars);

1008     if (rflag) {
1009         if (cflag && usefile != NULL) {
1010             /* Set the compression type */
1011             if (aflag)
1012                 detect_compress();

1014             if (jflag) {
1015                 compress_opt = compress_malloc(strlen(BZIP)
1016                     + 1);
1017                 (void) strcpy(compress_opt, BZIP);
1018             } else if (zflag) {
1019                 compress_opt = compress_malloc(strlen(GZIP)
1020                     + 1);
1021                 (void) strcpy(compress_opt, GZIP);
1022             } else if (Zflag) {
1023                 compress_opt =
1024                     compress_malloc(strlen(COMPRESS) + 1);
1025                 (void) strcpy(compress_opt, COMPRESS);
1026             } else if (Jflag) {
1027                 compress_opt = compress_malloc(strlen(XZ) + 1);
1028                 (void) strcpy(compress_opt, XZ);
1029             }
1030         } else {
1031             /*
1032              * Decompress if the file is compressed for
1033              * an update or replace.
1034              */
1035             if (strcmp(usefile, "-") != 0) {
1036                 check_compression();
1037                 if (compress_opt != NULL) {
1038                     decompress_file();
1039                 }
1040             }
1041         }
1043         if (cflag && tfile != NULL)
1044             usage();
1045         if (signal(SIGINT, SIG_IGN) != SIG_IGN)
1046             (void) signal(SIGINT, onintr);
1047         if (signal(SIGHUP, SIG_IGN) != SIG_IGN)
1048             (void) signal(SIGHUP, onhup);
1049         if (signal(SIGQUIT, SIG_IGN) != SIG_IGN)
1050             (void) signal(SIGQUIT, onquit);

```

```

1051             if (uflag) {
1052                 int tnum;
1053                 struct stat sbuf;

1055                 tmpdir = getenv("TMPDIR");
1056                 /*
1057                  * If the name is invalid or this isn't a directory,
1058                  * or the directory is not writable, then reset to
1059                  * a default temporary directory.
1060                  */
1061                 if (tmpdir == NULL || *tmpdir == '\0' ||
1062                     (strlen(tmpdir) + strlen(tmp_suffix)) > PATH_MAX) {
1063                     tmpdir = "/tmp";
1064                 } else if (stat(tmpdir, &sbuf) < 0 ||
1065                     (sbuf.st_mode & S_IFMT) != S_IFDIR ||
1066                     (sbuf.st_mode & S_IWRITE) == 0) {
1067                     tmpdir = "/tmp";
1068                 }
1069
1070                 if ((tname = calloc(1, strlen(tmpdir) +
1071                     strlen(tmp_suffix) + 1)) == NULL) {
1072                     v perror(1, gettext("tar: out of memory, "
1073                         "cannot create temporary file\n"));
1074                 }
1075                 (void) strcpy(tname, tmpdir);
1076                 (void) strcat(tname, tmp_suffix);

1077                 if ((tnum = mkstemp(tname)) == -1)
1078                     v perror(1, "%s", tname);
1079                 if ((tfile = fdopen(tnum, "w")) == NULL)
1080                     v perror(1, "%s", tname);
1081
1082                 if (strcmp(usefile, "-") == 0) {
1083                     if (cflag == 0)
1084                         fatal(gettext(
1085                             "can only create standard output archives."));
1086                     vfile = stderr;
1087                     mt = dup(1);
1088                     ++bflag;
1089                 } else {
1090                     if (cflag)
1091                         mt = open(usefile,
1092                             O_RDWR|O_CREAT|O_TRUNC, 0666);
1093                     else
1094                         mt = open(usefile, O_RDWR);
1095
1096                     if (mt < 0) {
1097                         if (cflag == 0 || (mt = creat(usefile, 0666))
1098                             < 0)
1099                             v perror(1, "%s", usefile);
1100                     }
1101
1102                     /* Get inode and device number of output file */
1103                     (void) fstat(mt, &stbuf);
1104                     mt_ino = stbuf.st_ino;
1105                     mt_dev = stbuf.st_dev;
1106                     mt_devtype = stbuf.st_mode & S_IFMT;
1107                     NotTape = !istape(mt, mt_devtype);
1108
1109                     if (rflag && !cflag && (mt_devtype == S_IFIFO))
1110                         fatal(gettext("cannot append to pipe or FIFO."));
1111
1112                     if (Aflag && vflag)
1113                         (void) printf(
1114                             gettext("Suppressing absolute pathnames\n"));
1115
1116

```

```
new/usr/src/cmd/tar/tar.c
1117         comp_pid = compress_file();
1118         dorep(argv);
1119         if (rflag && !cflag && (compress_opt != NULL))
1120             compress_back();
1121     } else if (xflag || tflag) {
1122         /*
1123          * for each argument, check to see if there is a "-I file" pair.
1124          * if so, move the 3rd argument into "-I"'s place, build_table()
1125          * using "file"'s name and increment argc one (the second
1126          * increment appears in the for loop) which removes the two
1127          * args "-I" and "file" from the argument vector.
1128          */
1129         for (argc = 0; argv[argc]; argc++) {
1130             if (strcmp(argv[argc], "-I") == 0) {
1131                 if (!argv[argc+1]) {
1132                     (void) fprintf(stderr, gettext(
1133                         "tar: missing argument for -I flag\n"));
1134                     done(2);
1135                 } else {
1136                     Iflag = 1;
1137                     argv[argc] = argv[argc+2];
1138                     build_table(include_tbl, argv[++argc]);
1139                 }
1140             } else if (strcmp(argv[argc], "-C") == 0) {
1141                 if (!argv[argc+1]) {
1142                     (void) fprintf(stderr, gettext("tar: "
1143                         "missing argument for -C flag\n"));
1144                     done(2);
1145                 } else if (xtract_chdir != NULL) {
1146                     (void) fprintf(stderr, gettext("tar: "
1147                         "extract should have only one -C "
1148                         "flag\n"));
1149                     done(2);
1150                 } else {
1151                     argv[argc] = argv[argc+2];
1152                     xtract_chdir = argv[++argc];
1153                 }
1154             }
1155         }
1156         if (strcmp(usefile, "-") == 0) {
1157             mt = dup(0);
1158             ++bflag;
1159             /* try to recover from short reads when reading stdin */
1160             ++Bflag;
1161         } else if ((mt = open(usefile, 0)) < 0)
1162             v perror(1, "%s", usefile);

1164     /* Decompress if the file is compressed */

1166     if (strcmp(usefile, "-") != 0) {
1167         check_compression();
1168         if (compress_opt != NULL)
1169             comp_pid = uncompress_file();
1170     }
1171     if (xflag) {
1172         if (xtract_chdir != NULL) {
1173             if (tar_chdir(xtract_chdir) < 0) {
1174                 v perror(1, gettext("can't change "
1175                         "directories to %s"), xtract_chdir);
1176             }
1177         }
1178         if (Aflag && vflag)
1179             (void) printf(gettext(
1180                         "Suppressing absolute pathnames.\n"));

1182         doxtract(argv);

```

```
15 new/usr/src/cmd/tar/tar.c

1183             } else if (tflag)
1184                     dotable(argv);
1185             }
1186         else
1187             usage();
1188
1189         done(Errflg);
1190
1191         /* Not reached: keep compiler quiet */
1192         return (1);
1193     }

1195 static boolean_t
1196 should_enable_debug(void)
1197 {
1198     const char *val;
1199     const char *truth[] = {
1200         "true",
1201         "1",
1202         "yes",
1203         "y",
1204         "please",
1205         NULL
1206     };
1207     unsigned int i;

1209     if ((val = getenv("DEBUG_TAR")) == NULL) {
1210         return (B_FALSE);
1211     }

1213     for (i = 0; truth[i] != NULL; i++) {
1214         if (strcmp(val, truth[i]) == 0) {
1215             return (B_TRUE);
1216         }
1217     }

1219     return (B_FALSE);
1220 }

1222 /*PRINTFLIKE1*/
1223 static void
1224 dlog(const char *format, ...)
1225 {
1226     va_list ap;

1228     if (!debug_output) {
1229         return;
1230     }

1232     va_start(ap, format);
1233     (void) fprintf(stderr, "tar: DEBUG: ");
1234     (void) vfprintf(stderr, format, ap);
1235     va_end(ap);
1236 }

1238 static void
1239 usage(void)
1240 {
1241     (void) fprintf(stderr, gettext(
1242 #if defined(O_XATTR)
1243 #if defined(_PC_SATTR_ENABLED)
1244         "Usage: tar {c|r|t|u|x}[BDeEFhilmnopPTvw@[0-7]][bf][X...]"
1245 #else
1246         "Usage: tar {c|r|t|u|x}[BDeEFhilmnopPTvw@[0-7]][bf][X...]"
1247 #endif /* _PC_SATTR_ENABLED */
1248 #else
```

```

1249         "Usage: tar {c|r|t|u|x}[BDeEFhilmnopPTvw[0-7]][bf][X...]" "
1250 #endif /* O_XATTR */
1251         "[z|J|z|Z]"
1252         "[blocksize] [tarfile] [size] [exclude-file...]"
1253         "[file | -I include-file | -C directory file]...\\n\"");
1254     done(1);
1255 }
unchanged_portion_omitted

1460 /*
1461 * endtape - check for tape at end
1462 *
1463 * endtape checks the entry in dblock.dbuf to see if its the
1464 * special EOT entry. Endtape is usually called after getdir().
1465 *
1466 * endtape used to call backtape; it no longer does, he who
1467 * wants it backed up must call backtape himself
1468 * RETURNS:      0 if not EOT, tape position unaffected
1469 *                1 if      EOT, tape position unaffected
1470 */

1472 static boolean_t
1473 endtape(void)
1474 {
1475     if (dblock.dbuf.name[0] != '\0') {
1476         /*
1477          * The name field is populated.
1478          */
1479     return (B_FALSE);
1480 }

1482     if (is_posix && dblock.dbuf.prefix[0] != '\0') {
1483         /*
1484          * This is a ustar/POSIX archive, and although the name
1485          * field is empty the prefix field is not.
1486          */
1487     return (B_FALSE);
1488 }

1489 dlog("endtape(): found null header; EOT\\n");
1490 return (B_TRUE);
1491 if (dblock.dbuf.name[0] == '\0') { /* null header = EOT */
1492     return (1);
1493 } else
1494     return (0);
1495 }

1496 /*
1497 * getdir - get directory entry from tar tape
1498 *
1499 * getdir reads the next tarblock off the tape and cracks
1500 * it as a directory. The checksum must match properly.
1501 *
1502 * If tfile is non-null getdir writes the file name and mod date
1503 * to tfile.
1504 */
1505 static void
1506 getdir(void)
1507 {
1508 #ifdef EUC
1509     static int warn_chksum_sign = 0;
1510 #endif /* EUC */

```

```

1512 top:
1513     readtape((char *)&dblock);
1514     if (dblock.dbuf.name[0] == '\0')
1515         return;
1516     sp = &stbuf;
1517     (void) sscanf(dblock.dbuf.mode, "%8lo", &Gen.g_mode);
1518     (void) sscanf(dblock.dbuf.uid, "%8lo", (ulong_t *)&Gen.g_uid);
1519     (void) sscanf(dblock.dbuf.gid, "%8lo", (ulong_t *)&Gen.g_gid);
1520     (void) sscanf(dblock.dbuf.size, "%12" FMT_off_t_o, &Gen.g_filesz);
1521     (void) sscanf(dblock.dbuf.mtime, "%12lo", (ulong_t *)&Gen.g_mtime);
1522     (void) sscanf(dblock.dbuf.chksum, "%8o", &Gen.g_cksum);
1523     (void) sscanf(dblock.dbuf.devmajor, "%8lo", &Gen.g_devmajor);
1524     (void) sscanf(dblock.dbuf.devminor, "%8lo", &Gen.g_devminor);

1526     is_posix = (strcmp(dblock.dbuf.magic, magic_type) == 0);

1528     sp->st_mode = Gen.g_mode;
1529     if (is_posix && (sp->st_mode & S_IFMT) == 0) {
1530         if (is_posix && (sp->st_mode & S_IFMT) == 0)
1531             switch (dblock.dbuf.typeflag) {
1532                 case '0':
1533                     case 0:
1534                     case _XATTR_HDRTYPE:
1535                         case '0': case 0: case _XATTR_HDRTYPE:
1536                             sp->st_mode |= S_IFREG;
1537                             break;
1538                         case '1': /* hard link */
1539                             sp->st_mode |= S_IFLNK;
1540                             break;
1541                         case '2':
1542                             sp->st_mode |= S_IFBLK;
1543                             break;
1544                         case '3':
1545                             sp->st_mode |= S_IFCHR;
1546                             break;
1547                         case '4':
1548                             sp->st_mode |= S_IFBLK;
1549                             break;
1550                         case '5':
1551                             sp->st_mode |= S_IFDIR;
1552                             break;
1553                         case '6':
1554                             sp->st_mode |= S_IFIFO;
1555                             break;
1556                         default:
1557                             if (convtoreg(Gen.g_filesz))
1558                                 sp->st_mode |= S_IFREG;
1559                             break;
1560             }
1561             if ((dblock.dbuf.typeflag == 'X') || (dblock.dbuf.typeflag == 'L')) {
1562                 xhdrflag = 1; /* Currently processing extended header */
1563             } else {
1564                 xhdrflag = 0;
1565             }
1566             sp->st_uid = Gen.g_uid;
1567             sp->st_gid = Gen.g_gid;
1568             sp->st_size = Gen.g_filesz;
1569             sp->st_mtime = Gen.g_mtime;
1570             cksum = Gen.g_cksum;
1571             if (dblock.dbuf.extno != '\0') { /* split file? */
1572                 extno = dblock.dbuf.extno;
1573                 extsize = Gen.g_filesz;
1574             }

```

```

1575         extotal = dblockdbuf.extotal;
1576     } else {
1577         extno = 0;      /* tell others file not split */
1578         extsize = 0;
1579         extotal = 0;
1580     }
1582 #ifdef EUC
1583     if (chksum != checksum(&dblock)) {
1584         if (chksum != checksum_signed(&dblock)) {
1585             (void) fprintf(stderr, gettext(
1586                 "tar: directory checksum error\n"));
1587             if (iflag) {
1588                 Errflg = 2;
1589                 goto top;
1590             }
1591             done(2);
1592         } else {
1593             if (! warn_chksum_sign) {
1594                 warn_chksum_sign = 1;
1595                 (void) fprintf(stderr, gettext(
1596                     "tar: warning: tar file made with signed checksum\n"));
1597             }
1598         }
1599     }
1600 #else
1601     if (chksum != checksum(&dblock)) {
1602         (void) fprintf(stderr, gettext(
1603             "tar: directory checksum error\n"));
1604         if (iflag) {
1605             Errflg = 2;
1606             goto top;
1607         }
1608         done(2);
1609     }
1610 #endif /* EUC */
1611     if (tfile != NULL && Xhdrflag == 0) {
1612         /*
1613          * If an extended header is present, then time is available
1614          * in nanoseconds in the extended header data, so set it.
1615          * Otherwise, give an invalid value so that checkupdate will
1616          * not test beyond seconds.
1617          */
1618         if ((xhdr_flg & _X_MTIME))
1619             sp->st_mtim.tv_nsec = Xtarhdr.x_mtime.tv_nsec;
1620         else
1621             sp->st_mtim.tv_nsec = -1;
1622
1623         if (xhdr_flg & _X_PATH)
1624             (void) fprintf(tfile, "%s %10ld.%9.9ld\n",
1625                           Xtarhdr.x_path, sp->st_mtim.tv_sec,
1626                           sp->st_mtim.tv_nsec);
1627         else
1628             (void) fprintf(tfile, "%.*s %10ld.%9.9ld\n",
1629                           NAMSIZ, dblockdbuf.name, sp->st_mtim.tv_sec,
1630                           sp->st_mtim.tv_nsec);
1631     }
1633 #if defined(O_XATTR)
1634     Hiddendir = 0;
1635     if (xattrp && dblockdbuf.typeflag == _XATTR_HDRTYPE) {
1636         if (xattrbadhead) {
1637             free(xattrhead);
1638             xattrp = NULL;
1639             xattr_linkp = NULL;
1640             xattrhead = NULL;

```

```

1641             } else {
1642                 char    *aname = basename(xattrpath);
1643                 size_t  xindex = aname - xattrpath;
1644
1645                 if (xattrpath[xindex] == '.' &&
1646                     xattrpath[xindex + 1] == '\0' &&
1647                     xattrp->h_typeflag == '5') {
1648                     Hiddendir = 1;
1649                     sp->st_mode =
1650                         (S_IFDIR | (sp->st_mode & POSIXMODES));
1651                 }
1652                 dblockdbuf.typeflag = xattrp->h_typeflag;
1653             }
1654         }
1655     }
1656
1657
1658
1659     /*
1660      * passtape - skip over a file on the tape
1661      *
1662      * passtape skips over the next data file on the tape.
1663      * The tape directory entry must be in dblockdbuf. This
1664      * routine just eats the number of blocks computed from the
1665      * directory size entry; the tape must be (logically) positioned
1666      * right after the directory info.
1667      */
1668
1669 static void
1670 passtape(void)
1671 {
1672     blkcnt_t blocks;
1673     char buf[TBLOCK];
1674
1675     /*
1676      * Print some debugging information about the directory entry
1677      * we are skipping over:
1678      */
1679     dlog("passtape: typeflag \'%c\"\n", dblockdbuf.typeflag);
1680     if (dblockdbuf.name[0] != '\0') {
1681         dlog("passtape: name \'%s\"\n", dblockdbuf.name);
1682     }
1683     if (is_posix & dblockdbuf.prefix[0] != '\0') {
1684         dlog("passtape: prefix \'%s\"\n", dblockdbuf.prefix);
1685     }
1686
1687     /*
1688      * Types link(1), sym-link(2), char special(3), blk special(4),
1689      * directory(5), and FIFO(6) do not have data blocks associated
1690      * with them so just skip reading the data block.
1691      */
1692     if (dblockdbuf.typeflag == '1' || dblockdbuf.typeflag == '2' ||
1693         dblockdbuf.typeflag == '3' || dblockdbuf.typeflag == '4' ||
1694         dblockdbuf.typeflag == '5' || dblockdbuf.typeflag == '6')
1695         return;
1696     blocks = TBLOCKS(stbuf.st_size);
1697
1698     dlog("passtape: block count %" FMT_blkcnt_t "\n", blocks);
1699
1700     /* if operating on disk, seek instead of reading */
1701     if (NotTape)
1702         seekdisk(blocks);
1703     else
1704         while (blocks-- > 0)
1705             readtape(buf);

```

```

1706 }
1707 unchanged_portion_omitted_
1708
1709 static int
1710 putfile(char *longname, char *shortname, char *parent, attr_data_t *attrinfo,
1711     int filetype, int lev, int symlink_lev)
1712 {
1713     int infile = -1; /* deliberately invalid */
1714     blkcnt_t blocks;
1715     char buf[PATH_MAX + 2]; /* Add trailing slash and null */
1716     char *bigbuf;
1717     int maxread;
1718     int hint; /* amount to write to get "in sync" */
1719     char filetmp[PATH_MAX + 1];
1720     char *cp;
1721     char *name;
1722     char *attrparent = NULL;
1723     char *longattrname = NULL;
1724     file_list_t *child = NULL;
1725     file_list_t *child_end = NULL;
1726     file_list_t *cptr;
1727     struct dirent *dp;
1728     DIR *dirp;
1729     int i;
1730     int split;
1731     int dirfd = -1;
1732     int rc = PUT_NOTAS_LINK;
1733     int archtype = 0;
1734     int rw_sysattr = 0;
1735     char newparent[PATH_MAX + MAXNAMLEN + 1];
1736     char *prefix = "";
1737     char *tmpbuf;
1738     char goodbuf[PRESIZ + 2];
1739     char junkbuf[MAXNAM+1];
1740     char *lastslash;
1741     int j;
1742     struct stat sbuf;
1743     int readlink_max;
1744
1745     (void) memset(goodbuf, '\0', sizeof(goodbuf));
1746     (void) memset(junkbuf, '\0', sizeof(junkbuf));
1747
1748     xhdr_flg = 0;
1749
1750     if (filetype == XATTR_FILE) {
1751         attrparent = attrinfo->attr_parent;
1752         longattrname = attrinfo->attr_path;
1753         dirfd = attrinfo->attr_parentfd;
1754         rw_sysattr = attrinfo->attr_rw_sysattr;
1755     } else {
1756         dirfd = open(".", O_RDONLY);
1757     }
1758
1759     if (dirfd == -1) {
1760         (void) fprintf(stderr, gettext(
1761             "tar: unable to open%directory %s%s%s\n"),
1762             (filetype == XATTR_FILE) ? gettext(" attribute ") : " ",
1763             (attrparent == NULL) ? "" : gettext("of attribute "),
1764             (attrparent == NULL) ? "" : attrparent,
1765             (attrparent == NULL) ? "" : gettext(" of "),
1766             (filetype == XATTR_FILE) ? longname : parent));
1767         goto out;
1768     }
1769
1770     if (lev > MAXLEV) {
1771         (void) fprintf(stderr,
1772

```

```

1880         gettext("tar: directory nesting too deep, %s not dumped\n"),
1881         longname);
1882         goto out;
1883     }
1884
1885     if (getstat(dirfd, longname, shortname, attrparent))
1886         goto out;
1887
1888     if (hflag) {
1889         /*
1890          * Catch nesting where a file is a symlink to its directory.
1891          */
1892         j = fstatat(dirfd, shortname, &sbuf, AT_SYMLINK_NOFOLLOW);
1893         if (S_ISLNK(sbuf.st_mode)) {
1894             if (symlink_lev++ >= MAXSYMLINKS) {
1895                 (void) fprintf(stderr, gettext(
1896                     "tar: %s: Number of symbolic links "
1897                     "encountered during path name traversal "
1898                     "exceeds MAXSYMLINKS\n"), longname);
1899             Errflg = 1;
1900             goto out;
1901         }
1902     }
1903 }
1904
1905 /*
1906  * Check if the input file is the same as the tar file we
1907  * are creating
1908  */
1909 if ((mt_ino == stbuf.st_ino) && (mt_dev == stbuf.st_dev)) {
1910     (void) fprintf(stderr, gettext(
1911         "tar: %s%s%s same as archive file\n"),
1912         rw_sysattr ? gettext("system ") : "",
1913         (longattrname == NULL) ? "" : gettext("attribute "),
1914         (longattrname == NULL) ? "" : longattrname,
1915         (longattrname == NULL) ? "" : gettext(" of "),
1916         longname);
1917     Errflg = 1;
1918     goto out;
1919 }
1920 /*
1921  * Check size limit - we can't archive files that
1922  * exceed TAR_OFFSET_MAX bytes because of header
1923  * limitations. Exclude file types that set
1924  * st_size to zero below because they take no
1925  * archive space to represent contents.
1926  */
1927 if ((stbuf.st_size > (off_t)TAR_OFFSET_MAX) &&
1928     !S_ISDIR(stbuf.st_mode) &&
1929     !S_ISCHR(stbuf.st_mode) &&
1930     !S_ISBLK(stbuf.st_mode) &&
1931     (Eflag == 0)) {
1932     (void) fprintf(stderr, gettext(
1933         "tar: %s%s%s too large to archive. "
1934         "Use E function modifier.\n"),
1935         rw_sysattr ? gettext("system ") : "",
1936         (longattrname == NULL) ? "" : gettext("attribute "),
1937         (longattrname == NULL) ? "" : longattrname,
1938         (longattrname == NULL) ? "" : gettext(" of "),
1939         longname);
1940     if (errflag)
1941         exitflag = 1;
1942     Errflg = 1;
1943     goto out;
1944 }

```

```

1946     if (tfile != NULL && checkupdate(longname) == 0) {
1947         goto out;
1948     }
1949     if (checkw('r', longname) == 0) {
1950         goto out;
1951     }
1953     if (Fflag &&
1954         checkf(longname, (stbuf.st_mode & S_IFMT) == S_IFDIR, Fflag) == 0)
1955         goto out;
1957     if (Xflag) {
1958         if (is_in_table(exclude_tbl, longname)) {
1959             if (vflag) {
1960                 (void) fprintf(vfile, gettext(
1961                     "a %s excluded\n"), longname);
1962             }
1963             goto out;
1964         }
1965     }
1967     /*
1968      * If the length of the fullname is greater than MAXNAM,
1969      * print out a message and return (unless extended headers are used,
1970      * in which case fullname is limited to PATH_MAX).
1971     */
1973     if (((split = (int)strlen(longname)) > MAXNAM) && (Eflag == 0)) ||
1974     (split > PATH_MAX)) {
1975         (void) fprintf(stderr, gettext(
1976             "tar: %s: file name too long\n"), longname);
1977         if (errflag)
1978             exitflag = 1;
1979         Errflg = 1;
1980         goto out;
1981     }
1983     /*
1984      * We split the fullname into prefix and name components if any one
1985      * of three conditions holds:
1986      *   -- the length of the fullname exceeds NAMSIZ,
1987      *   -- the length of the fullname equals NAMSIZ, and the shortname
1988      *       is less than NAMSIZ, (splitting in this case preserves
1989      *       compatibility with 5.6 and 5.5.1 tar), or
1990      *   -- the length of the fullname equals NAMSIZ, the file is a
1991      *       directory and we are not in POSIX-conformant mode (where
1992      *       trailing slashes are removed from directories).
1993     */
1994     if ((split > NAMSIZ) ||
1995     (split == NAMSIZ && strlen(shortname) < NAMSIZ) ||
1996     (split == NAMSIZ && S_ISDIR(stbuf.st_mode) && !Pflag)) {
1997         /*
1998          * Since path is limited to PRESIZ characters, look for the
1999          * last slash within PRESIZ + 1 characters only.
2000        */
2001     (void) strncpy(&goodbuf[0], longname, min(split, PRESIZ + 1));
2002     tmpbuf = goodbuf;
2003     lastslash = strchr(tmpbuf, '/');
2004     if (lastslash == NULL) {
2005         i = split;           /* Length of name */
2006         j = 0;               /* Length of prefix */
2007         goodbuf[0] = '\0';
2008     } else {
2009         *lastslash = '\0';    /* Terminate the prefix */
2010         j = strlen(tmpbuf);
2011         i = split - j - 1;

```

```

2012     }
2013     /*
2014      * If the filename is greater than NAMSIZ we can't
2015      * archive the file unless we are using extended headers.
2016     */
2017     if ((i > NAMSIZ) || (i == NAMSIZ && S_ISDIR(stbuf.st_mode) &&
2018         !Pflag)) {
2019         /*
2020          * Determine which (filename or path) is too long. */
2021         lastslash = strrchr(longname, '/');
2022         if (lastslash != NULL)
2023             i = strlen(lastslash + 1);
2024         if (Eflag > 0) {
2025             xhdr_flg |= _X_PATH;
2026             Xtarhdr.x_path = longname;
2027             if (i <= NAMSIZ)
2028                 (void) strcpy(junkbuf, lastslash + 1);
2029             else
2030                 (void) sprintf(junkbuf, "%llu",
2031                               xhdr_count + 1);
2032             if (split - i - 1 > PRESIZ)
2033                 (void) strcpy(goodbuf, xhdr_dirname);
2034         } else {
2035             if ((i > NAMSIZ) || (i == NAMSIZ &&
2036                 S_ISDIR(stbuf.st_mode) && !Pflag))
2037                 (void) fprintf(stderr, gettext(
2038                     "tar: %s: filename is greater than "
2039                     "%d\n"), lastslash == NULL ?
2040                         longname : lastslash + 1, NAMSIZ);
2041             else
2042                 (void) fprintf(stderr, gettext(
2043                     "tar: %s: prefix is greater than %d"
2044                     "\n"), longname, PRESIZ);
2045             if (errflag)
2046                 exitflag = 1;
2047             Errflg = 1;
2048             goto out;
2049         }
2050         (void) strncpy(&junkbuf[0], longname + j + 1,
2051                       strlen(longname + j + 1));
2052         name = junkbuf;
2053         prefix = goodbuf;
2054     } else {
2055         name = longname;
2056     }
2057     if (Aflag) {
2058         if ((prefix != NULL) && (*prefix != '\0'))
2059             while (*prefix == '/')
2060                 ++prefix;
2061         else
2062             while (*name == '/')
2063                 ++name;
2064     }
2066     switch (stbuf.st_mode & S_IFMT) {
2067     case S_IFDIR:
2068         stbuf.st_size = (off_t)0;
2069         blocks = TBLOCKS(stbuf.st_size);
2071         if (filetype != XATTR_FILE && Hiddendir == 0) {
2072             i = 0;
2073             cp = buf;
2074             while ((*cp++ = longname[i++]))
2075                 ;
2076             *--cp = '/';
2077             *++cp = 0;

```

```

2078     }
2079     if (!oflag) {
2080         tomodes(&stbuf);
2081         if (build_dblock(name, tchar, '5', filetype,
2082             &stbuf, stbuf.st_dev, prefix) != 0) {
2083             goto out;
2084         }
2085         if (!Pflag) {
2086             /*
2087             * Old archives require a slash at the end
2088             * of a directory name.
2089             *
2090             * XXX
2091             * If directory name is too long, will
2092             * slash overflow field?
2093             */
2094         if (strlen(name) > (unsigned)NAMSIZ-1) {
2095             (void) fprintf(stderr, gettext(
2096                 "tar: %s: filename is greater "
2097                 "than %d\n"), name, NAMSIZ);
2098             if (errflag)
2099                 exitflag = 1;
2100             Errflg = 1;
2101             goto out;
2102         } else {
2103             if (strlen(name) == (NAMSIZ - 1)) {
2104                 (void) memcpy(dblock.dbuf.name,
2105                     name, NAMSIZ);
2106                 dblock.dbuf.name[NAMSIZ-1]
2107                     = '/';
2108             } else
2109                 (void) sprintf(dblock.dbuf.name,
2110                     "%s/", name);
2111             /*
2112             * need to recalculate checksum
2113             * because the name changed.
2114             */
2115             (void) sprintf(dblock.dbuf.chksum,
2116                 "%07o", checksum(&dblock));
2117         }
2118     }
2119
2120     if (put_extra_attributes(longname, shortname,
2121         longattrname, prefix, filetype, '5') != 0)
2122         goto out;
2123
2124 #if defined(O_XATTR)
2125 /*
2126     * Reset header typeflag when archiving directory, since
2127     * build_dblock changed it on us.
2128     */
2129     if (filetype == XATTR_FILE) {
2130         dblock.dbuf.typeflag = _XATTR_HDRTYPE;
2131     } else {
2132         dblock.dbuf.typeflag = '5';
2133     }
2134
2135 #else
2136     dblock.dbuf.typeflag = '5';
2137
2138     (void) sprintf(dblock.dbuf.chksum, "%07o",
2139         checksum(&dblock));
2140
2141     (void) writetbuf((char *)&dblock, 1);
2142
2143 }

```

```

2144             if (vflag) {
2145                 if (NotTape) {
2146                     dlog("seek = %" FMT_blkcnt_t "K\n", K(tapepos));
2147                 }
2148             #ifdef DEBUG
2149                 if (NotTape)
2150                     DEBUG("seek = %" FMT_blkcnt_t "K\t", K(tapepos),
2151                         0);
2152             #endif
2153             if (filetype == XATTR_FILE && Hiddendir) {
2154                 (void) fprintf(vfile,
2155                     gettext("a %s attribute %s "),
2156                     longname, longattrname);
2157             } else {
2158                 (void) fprintf(vfile, "a %s/ ", longname);
2159             }
2160             if (NotTape) {
2161                 if (NotTape)
2162                     (void) fprintf(vfile, "%" FMT_blkcnt_t "K\n",
2163                         K(blocks));
2164             } else {
2165                 (void) fprintf(vfile, gettext("%" FMT_blkcnt_t
2166                         " tape blocks\n"), blocks);
2167             }
2168
2169             /*
2170             * If hidden dir then break now since xattrs_put() will do
2171             * the iterating of the directory.
2172             *
2173             * At the moment, there can only be system attributes on
2174             * attributes. There can be no attributes on attributes or
2175             * directories within the attributes hidden directory hierarchy.
2176             */
2177             if (filetype == XATTR_FILE)
2178                 break;
2179
2180             if (*shortname != '/')
2181                 (void) sprintf(newparent, "%s/%s", parent, shortname);
2182             else
2183                 (void) sprintf(newparent, "%s", shortname);
2184
2185             if (tar_chdir(shortname) < 0) {
2186                 v perror(0, "%s", newparent);
2187                 goto out;
2188             }
2189             if ((dirp = opendir(".")) == NULL) {
2190                 v perror(0, gettext(
2191                     "can't open directory %s"), longname);
2192                 if (tar_chdir(parent) < 0)
2193                     v perror(0, gettext("cannot change back?: %s"),
2194                         parent);
2195                 goto out;
2196             }
2197
2198             /*
2199             * Create a list of files (children) in this directory to avoid
2200             * having to perform telldir()/seekdir().
2201             */
2202             while ((dp = readdir(dirp)) != NULL && !term) {
2203                 if ((strcmp(".", dp->d_name) == 0) ||
2204                     (strcmp("../", dp->d_name) == 0))
2205                     continue;

```

```

2203
2204     if (((cptr = (file_list_t *)calloc(sizeof (char),
2205         sizeof (file_list_t))) == NULL) ||
2206         ((cptr->name = strdup(dp->d_name)) == NULL)) {
2207         vperror(1, gettext(
2208             "Insufficient memory for directory "
2209             "list entry %s\n"),
2210             newparent, dp->d_name);
2211     }
2212
2213     /* Add the file to the list */
2214     if (child == NULL) {
2215         child = cptr;
2216     } else {
2217         child_end->next = cptr;
2218     }
2219     child_end = cptr;
2220 }
2221 (void) closedir(dirp);
2222 /*
2223 * Archive each of the files in the current directory.
2224 * If a file is a directory, putfile() is called
2225 * recursively to archive the file hierarchy of the
2226 * directory before archiving the next file in the
2227 * current directory.
2228 */
2229 while ((child != NULL) && !term) {
2230     (void) strcpy(cp, child->name);
2231     archtype = putfile(buf, cp, newparent, NULL,
2232         NORMAL_FILE, lev + 1, symlink_lev);
2233
2234     if (!exitflag) {
2235         if ((atflag || saflag) &&
2236             (archtype == PUT_NOTAS_LINK)) {
2237             xattrs_put(buf, cp, newparent, NULL);
2238         }
2239     if (exitflag)
2240         break;
2241
2242     /* Free each child as we are done processing it. */
2243     cptr = child;
2244     child = child->next;
2245     free(cptr->name);
2246     free(cptr);
2247 }
2248 if ((child != NULL) && !term) {
2249     free_children(child);
2250 }
2251
2252 if (tar_chdir(parent) < 0) {
2253     vperror(0, gettext("cannot change back?: %s"), parent);
2254 }
2255
2256 break;
2257
2258 case S_IFLNK:
2259     readlink_max = NAMSIZ;
2260     if (stbuf.st_size > NAMSIZ) {
2261         if (Eflag > 0) {
2262             xhdr_flg |= _X_LINKPATH;
2263             readlink_max = PATH_MAX;
2264         } else {
2265             (void) fprintf(stderr, gettext(
2266                 "tar: %s: symbolic link too long\n"),
2267                 longname);

```

```

2268
2269     if (errflag)
2270         exitflag = 1;
2271     Errflg = 1;
2272     goto out;
2273 }
2274 */
2275 /* Sym-links need header size of zero since you
2276 * don't store any data for this type.
2277 */
2278 stbuf.st_size = (off_t)0;
2279 tomodes(&stbuf);
2280 i = readlink(shortname, filetmp, readlink_max);
2281 if (i < 0) {
2282     vperror(0, gettext(
2283         "can't read symbolic link %s"), longname);
2284     goto out;
2285 } else {
2286     filetmp[i] = 0;
2287 }
2288 if (vflag)
2289     (void) fprintf(vfile, gettext(
2290         "a %s symbolic link to %s\n"),
2291         longname, filetmp);
2292 if (xhdr_flg & _X_LINKPATH) {
2293     Xtarhdr.x_linkpath = filetmp;
2294     if (build_dblock(name, tchar, '2', filetype, &stbuf,
2295         stbuf.st_dev, prefix) != 0)
2296         goto out;
2297 } else
2298     if (build_dblock(name, filetmp, '2', filetype, &stbuf,
2299         stbuf.st_dev, prefix) != 0)
2300         goto out;
2301 (void) writetbuf((char *)&dblock, 1);
2302 /*
2303 * No acls for symlinks: mode is always 777
2304 * dont call write ancillary
2305 */
2306 rc = PUT_AS_LINK;
2307 break;
2308 case S_IFREG:
2309     if ((infile = openat(dirfd, shortname, 0)) < 0) {
2310         vperror(0, gettext("unable to open %s%s%s"), longname,
2311             rw_systat ? gettext(" system") : "",
2312             (filetype == XATTR_FILE) ?
2313                 gettext(" attribute ") : "",
2314             (filetype == XATTR_FILE) ? (longattrname == NULL) ?
2315                 shortname : longattrname : "");
2316     goto out;
2317 }
2318
2319 blocks = TBLOCKS(stbuf.st_size);
2320
2321 if (put_link(name, longname, shortname, longattrname,
2322     prefix, filetype, '1') == 0) {
2323     (void) close(infile);
2324     rc = PUT_AS_LINK;
2325     goto out;
2326 }
2327
2328 tomodes(&stbuf);
2329
2330 /* correctly handle end of volume */
2331 while (mulvol && tapepos + blocks + 1 > blocklim) {
2332     /* split if floppy has some room and file is large */
2333     if ((blocklim - tapepos) >= EXTMIN) &&
2334

```

```

2335             ((blocks + 1) >= blocklim/10)) {
2336                 splitfile(longname, infile,
2337                         name, prefix, filetype);
2338                 (void) close(dirfd);
2339                 (void) close(infile);
2340                 goto out;
2341         }
2342     newvol(); /* not worth it--just get new volume */
2343 }
2344 #ifdef DEBUG
2345 dlog("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2346 DEBUG("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2347         blocks);
2348 #endif
2349 if (build_dblock(name, tchar, '0', filetype,
2350         &stbuf, stbuf.st_dev, prefix) != 0) {
2351     goto out;
2352 }
2353 if (vflag) {
2354     if (NotTape) {
2355         dlog("seek = %" FMT_blkcnt_t "K\n", K(tapepos));
2356     }
2357 #ifdef DEBUG
2358     if (NotTape)
2359         DEBUG("seek = %" FMT_blkcnt_t "K\t", K(tapepos),
2360               0);
2361 #endif
2362     (void) fprintf(vfile, "a %s%s%s ", longname,
2363     rw_sysattr ? gettext(" system") : "",
2364     (filetype == XATTR_FILE) ? gettext(
2365         " attribute ") : "",
2366     (filetype == XATTR_FILE) ?
2367         longattrname : "");
2368     if (NotTape)
2369         (void) fprintf(vfile, "%" FMT_blkcnt_t "K\n",
2370                       K(blocks));
2371     else
2372         (void) fprintf(vfile,
2373                         gettext("%" FMT_blkcnt_t " tape blocks\n"),
2374                         blocks);
2375 }
2376 if (put_extra_attributes(longname, shortname, longattrname,
2377     prefix, filetype, '0') != 0)
2378     goto out;
2379 /*
2380 * No need to reset typeflag for extended attribute here, since
2381 * put_extra_attributes already set it and we haven't called
2382 * build_dblock().
2383 */
2384 (void) sprintf(dblockdbuf.chksum, "%07o", checksum(&dblock));
2385 hint = writetbuf((char *)&dblock, 1);
2386 maxread = max(min(stbuf.st_blksize, stbuf.st_size),
2387     (nblock * TBLOCK));
2388 if ((bigbuf = calloc((unsigned)maxread, sizeof (char))) == 0) {
2389     maxread = TBLOCK;
2390     bigbuf = buf;
2391 }
2392 while (((i = (int)
2393         read(infile, bigbuf, min((hint*TBLOCK), maxread))) > 0) &&
2394     blocks) {
2395     blkcnt_t nblk;
2396     nblk = ((i-1)/TBLOCK)+1;

```

```

2397         if (nblk > blocks)
2398             nblk = blocks;
2399         hint = writetbuf(bigbuf, nblk);
2400         blocks -= nblk;
2401     }
2402     (void) close(infile);
2403     if (bigbuf != buf)
2404         free(bigbuf);
2405     if (i < 0)
2406         v perror(0, gettext("Read error on %s"), longname);
2407     else if (blocks != 0 || i != 0) {
2408         (void) fprintf(stderr, gettext(
2409             "tar: %s: file changed size\n"), longname);
2410         if (errflag) {
2411             exitflag = 1;
2412             Errflg = 1;
2413         } else if (!Dflag) {
2414             Errflg = 1;
2415         }
2416     }
2417     putempty(blocks);
2418     break;
2419 case S_IFIFO:
2420     blocks = TBLOCKS(stbuf.st_size);
2421     stbuf.st_size = (off_t)0;
2422     if (put_link(name, longname, shortname, longattrname,
2423         prefix, filetype, '6') == 0) {
2424         rc = PUT_AS_LINK;
2425         goto out;
2426     }
2427     tomodes(&stbuf);
2428     while (mulvol && tapepos + blocks + 1 > blocklim) {
2429         if (((blocklim - tapepos) >= EXTMIN) &&
2430             ((blocks + 1) >= blocklim/10)) {
2431             splitfile(longname, infile, name,
2432                     prefix, filetype);
2433             (void) close(dirfd);
2434             (void) close(infile);
2435             goto out;
2436         }
2437         newvol();
2438     }
2439 dlog("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2440 DEBUG("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2441         blocks);
2442 #endif
2443 if (vflag) {
2444     if (NotTape) {
2445         dlog("seek = %" FMT_blkcnt_t "K\n", K(tapepos));
2446     }
2447 #ifdef DEBUG
2448     if (NotTape)
2449         DEBUG("seek = %" FMT_blkcnt_t "K\t", K(tapepos),
2450               0);
2451     else
2452         (void) fprintf(vfile, gettext("a %s %"
2453                                     FMT_blkcnt_t "K\n "), longname, K(blocks));
2454     else
2455         (void) fprintf(vfile, gettext(
2456             "a %s %" FMT_blkcnt_t " tape blocks\n"),
2457             longname, blocks);
2458 }

```

```

2449         }
2450     if (build_dblock(name, tchar, '6', filetype,
2451                     &stbuf, stbuf.st_dev, prefix) != 0)
2452         goto out;
2453
2454     if (put_extra_attributes(longname, shortname, longattrname,
2455                             prefix, filetype, '6') != 0)
2456         goto out;
2457
2458     (void) sprintf(dblockdbuf.chksum, "%07o", checksum(&dblock));
2459     dblockdbuf.typeflag = '6';
2460
2461     (void) writetbuf((char *)&dblock, 1);
2462     break;
2463
2464 case S_IFCHR:
2465     stbuf.st_size = (off_t)0;
2466     blocks = TBLOCKS(stbuf.st_size);
2467     if (put_link(name, longname, shortname, longattrname,
2468                 prefix, filetype, '3') == 0) {
2469         rc = PUT_AS_LINK;
2470         goto out;
2471     }
2472     tomodes(&stbuf);
2473
2474     while (mulvol && tapepos + blocks + 1 > blocklim) {
2475         if (((blocklim - tapepos) >= EXTMIN) &&
2476             ((blocks + 1) >= blocklim/10)) {
2477             splitfile(longname, infile, name,
2478                      prefix, filetype);
2479             (void) close(dirfd);
2480             goto out;
2481         }
2482         newvol();
2483     }
2484     dlog("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2485
2486 #ifdef DEBUG
2487     DEBUG("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2488           blocks);
2489 #endif
2490
2491     if (vflag) {
2492         if (NotTape) {
2493             dlog("seek = %" FMT_blkcnt_t "K\t", K(tapepos));
2494
2495 #ifdef DEBUG
2496         if (NotTape)
2497             DEBUG("seek = %" FMT_blkcnt_t "K\t", K(tapepos),
2498                  0);
2499         if (NotTape)
2500             (void) fprintf(vfile, gettext("a %s %"
2501                           FMT_blkcnt_t "K\n"), longname, K(blocks));
2502     } else {
2503         (void) fprintf(vfile, gettext("a %s %"
2504                           FMT_blkcnt_t " tape blocks\n"), longname,
2505                           blocks);
2506     }
2507
2508     if (build_dblock(name, tchar, '3',
2509                     filetype, &stbuf, stbuf.st_rdev, prefix) != 0)
2510         goto out;
2511
2512     if (put_extra_attributes(longname, shortname, longattrname,
2513                             prefix, filetype, '3') != 0)
2514         goto out;
2515
2516     (void) sprintf(dblockdbuf.chksum, "%07o", checksum(&dblock));
2517     dblockdbuf.typeflag = '3';
2518
2519     (void) writetbuf((char *)&dblock, 1);
2520     break;
2521
2522 case S_IFBLK:
2523     stbuf.st_size = (off_t)0;
2524     blocks = TBLOCKS(stbuf.st_size);
2525     if (put_link(name, longname, shortname, longattrname,
2526                 prefix, filetype, '4') == 0) {
2527         rc = PUT_AS_LINK;
2528         goto out;
2529     }
2530     newvol();
2531
2532 #ifdef DEBUG
2533     DEBUG("putfile: %s wants %" FMT_blkcnt_t " blocks\n", longname,
2534           blocks);
2535 #endif
2536     if (vflag) {
2537         if (NotTape) {
2538             dlog("seek = %" FMT_blkcnt_t "K\n", K(tapepos));
2539         }
2540
2541 #ifdef DEBUG
2542         if (NotTape)
2543             DEBUG("seek = %" FMT_blkcnt_t "K\t", K(tapepos),
2544                  0);
2545         if (NotTape)
2546             (void) fprintf(vfile, "a %s ", longname);
2547         if (NotTape)
2548             (void) fprintf(vfile, "%" FMT_blkcnt_t "K\n",
2549                           K(blocks));
2550     }
2551
2552     if (build_dblock(name, tchar, '4',
2553                     filetype, &stbuf, stbuf.st_rdev, prefix) != 0)
2554         goto out;
2555
2556     if (put_extra_attributes(longname, shortname, longattrname,
2557                             prefix, filetype, '4') != 0)
2558         goto out;
2559
2560     (void) sprintf(dblockdbuf.chksum, "%07o", checksum(&dblock));
2561     dblockdbuf.typeflag = '4';
2562
2563     (void) writetbuf((char *)&dblock, 1);
2564     break;
2565
2566 default:
2567     (void) fprintf(stderr, gettext(
2568         "tar: %s is not a file. Not dumped\n"), longname);
2569     if (errflag)
2570         exit(1);
2571
2572     if (vflag)
2573         dlog("tar: %s is not a file. Not dumped\n", longname);
2574
2575     if (errflag)
2576         exit(1);
2577
2578     if (vflag)
2579         dlog("tar: %s is not a file. Not dumped\n", longname);
2580
2581     if (errflag)
2582         exit(1);
2583
2584     if (vflag)
2585         dlog("tar: %s is not a file. Not dumped\n", longname);
2586
2587     if (errflag)
2588         exit(1);
2589
2590     if (vflag)
2591         dlog("tar: %s is not a file. Not dumped\n", longname);
2592
2593     if (errflag)
2594         exit(1);
2595
2596     if (vflag)
2597         dlog("tar: %s is not a file. Not dumped\n", longname);
2598
2599     if (errflag)
2600         exit(1);
2601
2602     if (vflag)
2603         dlog("tar: %s is not a file. Not dumped\n", longname);
2604
2605     if (errflag)
2606         exit(1);
2607
2608     if (vflag)
2609         dlog("tar: %s is not a file. Not dumped\n", longname);
2610
2611     if (errflag)
2612         exit(1);
2613
2614     if (vflag)
2615         dlog("tar: %s is not a file. Not dumped\n", longname);
2616
2617     if (errflag)
2618         exit(1);
2619
2620     if (vflag)
2621         dlog("tar: %s is not a file. Not dumped\n", longname);
2622
2623     if (errflag)
2624         exit(1);
2625
2626     if (vflag)
2627         dlog("tar: %s is not a file. Not dumped\n", longname);
2628
2629     if (errflag)
2630         exit(1);
2631
2632     if (vflag)
2633         dlog("tar: %s is not a file. Not dumped\n", longname);
2634
2635     if (errflag)
2636         exit(1);
2637
2638     if (vflag)
2639         dlog("tar: %s is not a file. Not dumped\n", longname);
2640
2641     if (errflag)
2642         exit(1);
2643
2644     if (vflag)
2645         dlog("tar: %s is not a file. Not dumped\n", longname);
2646
2647     if (errflag)
2648         exit(1);
2649
2650     if (vflag)
2651         dlog("tar: %s is not a file. Not dumped\n", longname);
2652
2653     if (errflag)
2654         exit(1);
2655
2656     if (vflag)
2657         dlog("tar: %s is not a file. Not dumped\n", longname);
2658
2659     if (errflag)
2660         exit(1);
2661
2662     if (vflag)
2663         dlog("tar: %s is not a file. Not dumped\n", longname);
2664
2665     if (errflag)
2666         exit(1);
2667
2668     if (vflag)
2669         dlog("tar: %s is not a file. Not dumped\n", longname);
2670
2671     if (errflag)
2672         exit(1);
2673
2674     if (vflag)
2675         dlog("tar: %s is not a file. Not dumped\n", longname);
2676
2677     if (errflag)
2678         exit(1);
2679
2680     if (vflag)
2681         dlog("tar: %s is not a file. Not dumped\n", longname);
2682
2683     if (errflag)
2684         exit(1);
2685
2686     if (vflag)
2687         dlog("tar: %s is not a file. Not dumped\n", longname);
2688
2689     if (errflag)
2690         exit(1);
2691
2692     if (vflag)
2693         dlog("tar: %s is not a file. Not dumped\n", longname);
2694
2695     if (errflag)
2696         exit(1);
2697
2698     if (vflag)
2699         dlog("tar: %s is not a file. Not dumped\n", longname);
2700
2701     if (errflag)
2702         exit(1);
2703
2704     if (vflag)
2705         dlog("tar: %s is not a file. Not dumped\n", longname);
2706
2707     if (errflag)
2708         exit(1);
2709
2710     if (vflag)
2711         dlog("tar: %s is not a file. Not dumped\n", longname);
2712
2713     if (errflag)
2714         exit(1);
2715
2716     if (vflag)
2717         dlog("tar: %s is not a file. Not dumped\n", longname);
2718
2719     if (errflag)
2720         exit(1);
2721
2722     if (vflag)
2723         dlog("tar: %s is not a file. Not dumped\n", longname);
2724
2725     if (errflag)
2726         exit(1);
2727
2728     if (vflag)
2729         dlog("tar: %s is not a file. Not dumped\n", longname);
2730
2731     if (errflag)
2732         exit(1);
2733
2734     if (vflag)
2735         dlog("tar: %s is not a file. Not dumped\n", longname);
2736
2737     if (errflag)
2738         exit(1);
2739
2740     if (vflag)
2741         dlog("tar: %s is not a file. Not dumped\n", longname);
2742
2743     if (errflag)
2744         exit(1);
2745
2746     if (vflag)
2747         dlog("tar: %s is not a file. Not dumped\n", longname);
2748
2749     if (errflag)
2750         exit(1);
2751
2752     if (vflag)
2753         dlog("tar: %s is not a file. Not dumped\n", longname);
2754
2755     if (errflag)
2756         exit(1);
2757
2758     if (vflag)
2759         dlog("tar: %s is not a file. Not dumped\n", longname);
2760
2761     if (errflag)
2762         exit(1);
2763
2764     if (vflag)
2765         dlog("tar: %s is not a file. Not dumped\n", longname);
2766
2767     if (errflag)
2768         exit(1);
2769
2770     if (vflag)
2771         dlog("tar: %s is not a file. Not dumped\n", longname);
2772
2773     if (errflag)
2774         exit(1);
2775
2776     if (vflag)
2777         dlog("tar: %s is not a file. Not dumped\n", longname);
2778
2779     if (errflag)
2780         exit(1);
2781
2782     if (vflag)
2783         dlog("tar: %s is not a file. Not dumped\n", longname);
2784
2785     if (errflag)
2786         exit(1);
2787
2788     if (vflag)
2789         dlog("tar: %s is not a file. Not dumped\n", longname);
2790
2791     if (errflag)
2792         exit(1);
2793
2794     if (vflag)
2795         dlog("tar: %s is not a file. Not dumped\n", longname);
2796
2797     if (errflag)
2798         exit(1);
2799
2800     if (vflag)
2801         dlog("tar: %s is not a file. Not dumped\n", longname);
2802
2803     if (errflag)
2804         exit(1);
2805
2806     if (vflag)
2807         dlog("tar: %s is not a file. Not dumped\n", longname);
2808
2809     if (errflag)
2810         exit(1);
2811
2812     if (vflag)
2813         dlog("tar: %s is not a file. Not dumped\n", longname);
2814
2815     if (errflag)
2816         exit(1);
2817
2818     if (vflag)
2819         dlog("tar: %s is not a file. Not dumped\n", longname);
2820
2821     if (errflag)
2822         exit(1);
2823
2824     if (vflag)
2825         dlog("tar: %s is not a file. Not dumped\n", longname);
2826
2827     if (errflag)
2828         exit(1);
2829
2830     if (vflag)
2831         dlog("tar: %s is not a file. Not dumped\n", longname);
2832
2833     if (errflag)
2834         exit(1);
2835
2836     if (vflag)
2837         dlog("tar: %s is not a file. Not dumped\n", longname);
2838
2839     if (errflag)
2840         exit(1);
2841
2842     if (vflag)
2843         dlog("tar: %s is not a file. Not dumped\n", longname);
2844
2845     if (errflag)
2846         exit(1);
2847
2848     if (vflag)
2849         dlog("tar: %s is not a file. Not dumped\n", longname);
2850
2851     if (errflag)
2852         exit(1);
2853
2854     if (vflag)
2855         dlog("tar: %s is not a file. Not dumped\n", longname);
2856
2857     if (errflag)
2858         exit(1);
2859
2860     if (vflag)
2861         dlog("tar: %s is not a file. Not dumped\n", longname);
2862
2863     if (errflag)
2864         exit(1);
2865
2866     if (vflag)
2867         dlog("tar: %s is not a file. Not dumped\n", longname);
2868
2869     if (errflag)
2870         exit(1);
2871
2872     if (vflag)
2873         dlog("tar: %s is not a file. Not dumped\n", longname);
2874
2875     if (errflag)
2876         exit(1);
2877
2878     if (vflag)
2879         dlog("tar: %s is not a file. Not dumped\n", longname);
2880
2881     if (errflag)
2882         exit(1);
2883
2884     if (vflag)
2885         dlog("tar: %s is not a file. Not dumped\n", longname);
2886
2887     if (errflag)
2888         exit(1);
2889
2890     if (vflag)
2891         dlog("tar: %s is not a file. Not dumped\n", longname);
2892
2893     if (errflag)
2894         exit(1);
2895
2896     if (vflag)
2897         dlog("tar: %s is not a file. Not dumped\n", longname);
2898
2899     if (errflag)
2900         exit(1);
2901
2902     if (vflag)
2903         dlog("tar: %s is not a file. Not dumped\n", longname);
2904
2905     if (errflag)
2906         exit(1);
2907
2908     if (vflag)
2909         dlog("tar: %s is not a file. Not dumped\n", longname);
2910
2911     if (errflag)
2912         exit(1);
2913
2914     if (vflag)
2915         dlog("tar: %s is not a file. Not dumped\n", longname);
2916
2917     if (errflag)
2918         exit(1);
2919
2920     if (vflag)
2921         dlog("tar: %s is not a file. Not dumped\n", longname);
2922
2923     if (errflag)
2924         exit(1);
2925
2926     if (vflag)
2927         dlog("tar: %s is not a file. Not dumped\n", longname);
2928
2929     if (errflag)
2930         exit(1);
2931
2932     if (vflag)
2933         dlog("tar: %s is not a file. Not dumped\n", longname);
2934
2935     if (errflag)
2936         exit(1);
2937
2938     if (vflag)
2939         dlog("tar: %s is not a file. Not dumped\n", longname);
2940
2941     if (errflag)
2942         exit(1);
2943
2944     if (vflag)
2945         dlog("tar: %s is not a file. Not dumped\n", longname);
2946
2947     if (errflag)
2948         exit(1);
2949
2950     if (vflag)
2951         dlog("tar: %s is not a file. Not dumped\n", longname);
2952
2953     if (errflag)
2954         exit(1);
2955
2956     if (vflag)
2957         dlog("tar: %s is not a file. Not dumped\n", longname);
2958
2959     if (errflag)
2960         exit(1);
2961
2962     if (vflag)
2963         dlog("tar: %s is not a file. Not dumped\n", longname);
2964
2965     if (errflag)
2966         exit(1);
2967
2968     if (vflag)
2969         dlog("tar: %s is not a file. Not dumped\n", longname);
2970
2971     if (errflag)
2972         exit(1);
2973
2974     if (vflag)
2975         dlog("tar: %s is not a file. Not dumped\n", longname);
2976
2977     if (errflag)
2978         exit(1);
2979
2980     if (vflag)
2981         dlog("tar: %s is not a file. Not dumped\n", longname);
2982
2983     if (errflag)
2984         exit(1);
2985
2986     if (vflag)
2987         dlog("tar: %s is not a file. Not dumped\n", longname);
2988
2989     if (errflag)
2990         exit(1);
2991
2992     if (vflag)
2993         dlog("tar: %s is not a file. Not dumped\n", longname);
2994
2995     if (errflag)
2996         exit(1);
2997
2998     if (vflag)
2999         dlog("tar: %s is not a file. Not dumped\n", longname);
2999
3000     if (errflag)
3001         exit(1);
3002
3003     if (vflag)
3004         dlog("tar: %s is not a file. Not dumped\n", longname);
3005
3006     if (errflag)
3007         exit(1);
3008
3009     if (vflag)
3010         dlog("tar: %s is not a file. Not dumped\n", longname);
3011
3012     if (errflag)
3013         exit(1);
3014
3015     if (vflag)
3016         dlog("tar: %s is not a file. Not dumped\n", longname);
3017
3018     if (errflag)
3019         exit(1);
3020
3021     if (vflag)
3022         dlog("tar: %s is not a file. Not dumped\n", longname);
3023
3024     if (errflag)
3025         exit(1);
3026
3027     if (vflag)
3028         dlog("tar: %s is not a file. Not dumped\n", longname);
3029
3030     if (errflag)
3031         exit(1);
3032
3033     if (vflag)
3034         dlog("tar: %s is not a file. Not dumped\n", longname);
3035
3036     if (errflag)
3037         exit(1);
3038
3039     if (vflag)
3040         dlog("tar: %s is not a file. Not dumped\n", longname);
3041
3042     if (errflag)
3043         exit(1);
3044
3045     if (vflag)
3046         dlog("tar: %s is not a file. Not dumped\n", longname);
3047
3048     if (errflag)
3049         exit(1);
3050
3051     if (vflag)
3052         dlog("tar: %s is not a file. Not dumped\n", longname);
3053
3054     if (errflag)
3055         exit(1);
3056
3057     if (vflag)
3058         dlog("tar: %s is not a file. Not dumped\n", longname);
3059
3060     if (errflag)
3061         exit(1);
3062
3063     if (vflag)
3064         dlog("tar: %s is not a file. Not dumped\n", longname);
3065
3066     if (errflag)
3067         exit(1);
3068
3069     if (vflag)
3070         dlog("tar: %s is not a file. Not dumped\n", longname);
3071
3072     if (errflag)
3073         exit(1);
3074
3075     if (vflag)
3076         dlog("tar: %s is not a file. Not dumped\n", longname);
3077
3078     if (errflag)
3079         exit(1);
3080
3081     if (vflag)
3082         dlog("tar: %s is not a file. Not dumped\n", longname);
3083
3084     if (errflag)
3085         exit(1);
3086
3087     if (vflag)
3088         dlog("tar: %s is not a file. Not dumped\n", longname);
3089
3090     if (errflag)
3091         exit(1);
3092
3093     if (vflag)
3094         dlog("tar: %s is not a file. Not dumped\n", longname);
3095
3096     if (errflag)
3097         exit(1);
3098
3099     if (vflag)
3100         dlog("tar: %s is not a file. Not dumped\n", longname);
3101
3102     if (errflag)
3103         exit(1);
3104
3105     if (vflag)
3106         dlog("tar: %s is not a file. Not dumped\n", longname);
3107
3108     if (errflag)
3109         exit(1);
3110
3111     if (vflag)
3112         dlog("tar: %s is not a file. Not dumped\n", longname);
3113
3114     if (errflag)
3115         exit(1);
3116
3117     if (vflag)
3118         dlog("tar: %s is not a file. Not dumped\n", longname);
3119
3120     if (errflag)
3121         exit(1);
3122
3123     if (vflag)
3124         dlog("tar: %s is not a file. Not dumped\n", longname);
3125
3126     if (errflag)
3127         exit(1);
3128
3129     if (vflag)
3130         dlog("tar: %s is not a file. Not dumped\n", longname);
3131
3132     if (errflag)
3133         exit(1);
3134
3135     if (vflag)
3136         dlog("tar: %s is not a file. Not dumped\n", longname);
3137
3138     if (errflag)
3139         exit(1);
3140
3141     if (vflag)
3142         dlog("tar: %s is not a file. Not dumped\n", longname);
3143
3144     if (errflag)
3145         exit(1);
3146
3147     if (vflag)
3148         dlog("tar: %s is not a file. Not dumped\n", longname);
3149
3150     if (errflag)
3151         exit(1);
3152
3153     if (vflag)
3154         dlog("tar: %s is not a file. Not dumped\n", longname);
3155
3156     if (errflag)
3157         exit(1);
3158
3159     if (vflag)
3160         dlog("tar: %s is not a file. Not dumped\n", longname);
3161
3162     if (errflag)
3163         exit(1);
3164
3165     if (vflag)
3166         dlog("tar: %s is not a file. Not dumped\n", longname);
3167
3168     if (errflag)
3169         exit(1);
3170
3171     if (vflag)
3172         dlog("tar: %s is not a file. Not dumped\n", longname);
3173
3174     if (errflag)
3175         exit(1);
3176
3177     if (vflag)
3178         dlog("tar: %s is not a file. Not dumped\n", longname);
3179
3180     if (errflag)
3181         exit(1);
3182
3183     if (vflag)
3184         dlog("tar: %s is not a file. Not dumped\n", longname);
3185
3186     if (errflag)
3187         exit(1);
3188
3189     if (vflag)
3190         dlog("tar: %s is not a file. Not dumped\n", longname);
3191
3192     if (errflag)
3193         exit(1);
3194
3195     if (vflag)
3196         dlog("tar: %s is not a file. Not dumped\n", longname);
3197
3198     if (errflag)
3199         exit(1);
3200
3201     if (vflag)
3202         dlog("tar: %s is not a file. Not dumped\n", longname);
3203
3204     if (errflag)
3205         exit(1);
3206
3207     if (vflag)
3208         dlog("tar: %s is not a file. Not dumped\n", longname);
3209
3210     if (errflag)
3211         exit(1);
3212
3213     if (vflag)
3214         dlog("tar: %s is not a file. Not dumped\n", longname);
3215
3216     if (errflag)
3217         exit(1);
3218
3219     if (vflag)
3220         dlog("tar: %s is not a file. Not dumped\n", longname);
3221
3222     if (errflag)
3223         exit(1);
3224
3225     if (vflag)
3226         dlog("tar: %s is not a file. Not dumped\n", longname);
3227
3228     if (errflag)
3229         exit(1);
3230
3231     if (vflag)
3232         dlog("tar: %s is not a file. Not dumped\n", longname);
3233
3234     if (errflag)
3235         exit(1);
3236
3237     if (vflag)
3238         dlog("tar: %s is not a file. Not dumped\n", longname);
3239
3240     if (errflag)
3241         exit(1);
3242
3243     if (vflag)
3244         dlog("tar: %s is not a file. Not dumped\n", longname);
3245
3246     if (errflag)
3247         exit(1);
3248
3249     if (vflag)
3250         dlog("tar: %s is not a file. Not dumped\n", longname);
3251
3252     if (errflag)
3253         exit(1);
3254
3255     if (vflag)
3256         dlog("tar: %s is not a file. Not dumped\n", longname);
3257
3258     if (errflag)
3259         exit(1);
3260
3261     if (vflag)
3262         dlog("tar: %s is not a file. Not dumped\n", longname);
3263
3264     if (errflag)
3265         exit(1);
3266
3267     if (vflag)
3268         dlog("tar: %s is not a file. Not dumped\n", longname);
3269
3270     if (errflag)
3271         exit(1);
3272
3273     if (vflag)
3274         dlog("tar: %s is not a file. Not dumped\n", longname);
3275
3276     if (errflag)
3277         exit(1);
3278
3279     if (vflag)
3280         dlog("tar: %s is not a file. Not dumped\n", longname);
3281
3282     if (errflag)
3283         exit(1);
3284
3285     if (vflag)
3286         dlog("tar: %s is not a file. Not dumped\n", longname);
3287
3288     if (errflag)
3289         exit(1);
3290
3291     if (vflag)
3292         dlog("tar: %s is not a file. Not dumped\n", longname);
3293
3294     if (errflag)
3295         exit(1);
3296
3297     if (vflag)
3298         dlog("tar: %s is not a file. Not dumped\n", longname);
3299
3300     if (errflag)
3301         exit(1);
3302
3303     if (vflag)
3304         dlog("tar: %s is not a file. Not dumped\n", longname);
3305
3306     if (errflag)
3307         exit(1);
3308
3309     if (vflag)
3310         dlog("tar: %s is not a file. Not dumped\n", longname);
3311
3312     if (errflag)
3313         exit(1);
3314
3315     if (vflag)
3316         dlog("tar: %s is not a file. Not dumped\n", longname);
3317
3318     if (errflag)
3319         exit(1);
3320
3321     if (vflag)
3322         dlog("tar: %s is not a file. Not dumped\n", longname);
3323
3324     if (errflag)
3325         exit(1);
3326
3327     if (vflag)
3328         dlog("tar: %s is not a file. Not dumped\n", longname);
3329
3330     if (errflag)
3331         exit(1);
3332
3333     if (vflag)
3334         dlog("tar: %s is not a file. Not dumped\n", longname);
3335
3336     if (errflag)
3337         exit(1);
3338
3339     if (vflag)
3340         dlog("tar: %s is not a file. Not dumped\n", longname);
3341
3342     if (errflag)
3343         exit(1);
3344
3345     if (vflag)
3346         dlog("tar: %s is not a file. Not dumped\n", longname);
3347
3348     if (errflag)
3349         exit(1);
3350
3351     if (vflag)
3352         dlog("tar: %s is not a file. Not dumped\n", longname);
3353
3354     if (errflag)
3355         exit(1);
3356
3357     if (vflag)
3358         dlog("tar: %s is not a file. Not dumped\n", longname);
3359
3360     if (errflag)
3361         exit(1);
3362
3363     if (vflag)
3364         dlog("tar: %s is not a file. Not dumped\n", longname);
3365
3366     if (errflag)
3367         exit(1);
3368
3369     if (vflag)
3370         dlog("tar: %s is not a file. Not dumped\n", longname);
3371
3372     if (errflag)
3373         exit(1);
3374
3375     if (vflag)
3376         dlog("tar: %s is not a file. Not dumped\n", longname);
3377
3378     if (errflag)
3379         exit(1);
3380
3381     if (vflag)
3382         dlog("tar: %s is not a file. Not dumped\n", longname);
3383
3384     if (errflag)
3385         exit(1);
3386
3387     if (vflag)
3388         dlog("tar: %s is not a file. Not dumped\n", longname);
3389
3390     if (errflag)
3391         exit(1);
3392
3393     if (vflag)
3394         dlog("tar: %s is not a file. Not dumped\n", longname);
3395
3396     if (errflag)
3397         exit(1);
3398
3399     if (vflag)
3400         dlog("tar: %s is not a file. Not dumped\n", longname);
3401
3402     if (errflag)
3403         exit(1);
3404
3405     if (vflag)
3406         dlog("tar: %s is not a file. Not dumped\n", longname);
3407
3408     if (errflag)
3409         exit(1);
3410
3411     if (vflag)
3412         dlog("tar: %s is not a file. Not dumped\n", longname);
3413
3414     if (errflag)
3415         exit(1);
3416
3417     if (vflag)
3418         dlog("tar: %s is not a file. Not dumped\n", longname);
3419
3420     if (errflag)
3421         exit(1);
3422
3423     if (vflag)
3424         dlog("tar: %s is not a file. Not dumped\n", longname);
3425
3426     if (errflag)
3427         exit(1);
3428
3429     if (vflag)
3430         dlog("tar: %s is not a file. Not dumped\n", longname);
3431
3432     if (errflag)
3433         exit(1);
3434
3435     if (vflag)
3436         dlog("tar: %s is not a file. Not dumped\n", longname);
3437
3438     if (errflag)
3439         exit(1);
3440
3441     if (vflag)
3442         dlog("tar: %s is not a file. Not dumped\n", longname);
3443
3444     if (errflag)
3445         exit(1);
3446
3447     if (vflag)
3448         dlog("tar: %s is not a file. Not dumped\n", longname);
3449
3450     if (errflag)
3451         exit(1);
3452
3453     if (vflag)
3454         dlog("tar: %s is not a file. Not dumped\n", longname);
3455
3456     if (errflag)
3457         exit(1);
3458
3459     if (vflag)
3460         dlog("tar: %s is not a file. Not dumped\n", longname);
3461
3462     if (errflag)
3463         exit(1);
3464
3465     if (vflag)
3466         dlog("tar: %s is not a file. Not dumped\n", longname);
3467
3468     if (errflag)
3469         exit(1);
3470
3471     if (vflag)
3472         dlog("tar: %s is not a file. Not dumped\n", longname);
3473
3474     if (errflag)
3475         exit(1);
3476
3477     if (vflag)
3478         dlog("tar: %s is not a file. Not dumped\n", longname);
3479
3480     if (errflag)
3481         exit(1);
3482
3483     if (vflag)
3484         dlog("tar: %s is not a file. Not dumped\n", longname);
3485
3486     if (errflag)
3487         exit(1);
3488
3489     if (vflag)
3490         dlog("tar: %s is not a file. Not dumped\n", longname);
3491
3492     if (errflag)
3493         exit(1);
3494
3495     if (vflag)
3496         dlog("tar: %s is not a file. Not dumped\n", longname);
3497
3498     if (errflag
```

```

2563           exitflag = 1;
2564           Errflg = 1;
2565           goto out;
2566       }
2567
2568 out:
2569     if ((dirfd != -1) && (filetype != XATTR_FILE)) {
2570         (void) close(dirfd);
2571     }
2572     return (rc);
2573 }
unchanged_portion_omitted

4835 /*
4836 *    newvol  get new floppy (or tape) volume
4837 *
4838 *    newvol();           resets tapepos and first to TRUE, prompts for
4839 *                        for new volume, and waits.
4840 *    if dumping, end-of-file is written onto the tape.
4841 */
4842 static void
4843 newvol(void)
4844 {
4845     int c;
4846
4847     if (dumping) {
4848         dlog("newvol called with 'dumping' set\n");
4849 #ifdef DEBUG
4850         DEBUG("newvol called with 'dumping' set\n", 0, 0);
4851 #endif
4852         putempty((blkcnt_t)2); /* 2 EOT marks */
4853         closevol();
4854         flushape();
4855         sync();
4856         tapepos = 0;
4857     } else
4858         first = TRUE;
4859     if (close(mt) != 0)
4860         v perror(2, gettext("close error"));
4861     mt = 0;
4862     (void) fprintf(stderr, gettext(
4863         "tar: \\007please insert new volume, then press RETURN."));
4864     (void) fseek(stdin, (off_t)0, 2); /* scan over read-ahead */
4865     while ((c = getchar()) != '\n' && ! term)
4866         if (c == EOF)
4867             done(Errflg);
4868     if (term)
4869         done(Errflg);
4870
4871     errno = 0;
4872
4873     if (strcmp(usefile, "-") == 0) {
4874         mt = dup(1);
4875     } else {
4876         mt = open(usefile, dumping ? update : 0);
4877     }
4878
4879     if (mt < 0) {
4880         (void) fprintf(stderr, gettext(
4881             "tar: cannot reopen %s (%s)\n",
4882             dumping ? gettext("output") : gettext("input"), usefile));
4883
4884         dlog("update=%d, usefile=%s ", update, usefile);
4885         dlog("mt=%d, [%s]\n", mt, strerror(errno));

```

```

4828 #ifdef DEBUG
4829     DEBUG("update=%d, usefile=%s ", update, usefile);
4830     DEBUG("mt=%d, [%s]\n", mt, strerror(errno));
4831 #endif
4832
4833         done(2);
4834     }
4835 }
unchanged_portion_omitted

5045 /*
5046 *    seekdisk      seek to next file on archive
5047 *
5048 *    called by passtape() only
5049 *
5050 *    WARNING: expects "nblock" to be set, that is, readtape() to have
5051 *              already been called. Since passtape() is only called
5052 *              after a file header block has been read (why else would
5053 *              we skip to next file?), this is currently safe.
5054 *
5055 *    changed to guarantee SYS_BLOCK boundary
5056 */
5057
5058 static void
5059 seekdisk(blkcnt_t blocks)
5060 {
5061     off_t seekval;
5062 #if SYS_BLOCK > TBLOCK
5063     /* handle non-multiple of SYS_BLOCK */
5064     blkcnt_t nxb; /* # extra blocks */
5065 #endif
5066
5067     tapepos += blocks;
5068     dlog("%" FMT_blkcnt_t " called\n", blocks);
5069 #ifdef DEBUG
5070     DEBUG("%" FMT_blkcnt_t " called\n", blocks, 0);
5071 #endif
5072     if (recono + blocks <= nblock) {
5073         recono += blocks;
5074         return;
5075     }
5076     if (recono > nblock)
5077         recono = nblock;
5078     seekval = (off_t)blocks - (nblock - recono);
5079     recono = nblock; /* so readline() reads next time through */
5080 #if SYS_BLOCK > TBLOCK
5081     nxb = (blkcnt_t)(seekval % (off_t)(SYS_BLOCK / TBLOCK));
5082     dlog("xtrablk=%" FMT_blkcnt_t " seekval=%" FMT_blkcnt_t " blks\n",
5083          nxb, seekval);
5084 #endif
5085     if (nxb && nxb > seekval) /* don't seek--we'll read */
5086         goto noseek;
5087     seekval -= nxb; /* don't seek quite so far */
5088 #endif
5089     if (lseek(mt, (off_t)(TBLOCK * seekval), 1) == (off_t)-1) {
5090         (void) fprintf(stderr, gettext(
5091             "tar: device seek error\n"));
5092         done(3);
5093     }
5094 #if SYS_BLOCK > TBLOCK
5095     /* read those extra blocks */
5096     if (nxb) {

```

```

5094     dlog("reading extra blocks\n", 0, 0);
5046 #ifdef DEBUG      DEBUG("reading extra blocks\n", 0, 0);
5047 #endif
5048     if (read(mt, tbuf, TBLOCK*nblock) < 0) {
5049         (void) fprintf(stderr, gettext(
5050             "tar: read error while skipping file\n"));
5051         done(8);
5052     }
5053     recno = nxb; /* so we don't read in next readtape() */
5054 }
5102 #endif
5103 }



---


unchanged portion omitted

5247 /*
5248 *      backtape - reposition tape after reading soft "EOF" record
5249 *
5250 *      Backtape tries to reposition the tape back over the EOF
5251 *      record. This is for the 'u' and 'r' function letters so that the
5252 *      tape can be extended. This code is not well designed, but
5253 *      I'm confident that the only callers who care about the
5254 *      backspace-over-EOF feature are those involved in 'u' and 'r'.
5255 *
5256 *      The proper way to backup the tape is through the use of mtio.
5257 *      Earlier spins used lseek combined with reads in a confusing
5258 *      maneuver that only worked on 4.x, but shouldn't have, even
5259 *      there. Lseeks are explicitly not supported for tape devices.
5260 */

5262 static void
5263 backtape(void)
5264 {
5265     struct mttop mtcmd;
5266     dlog("backtape() called, recno=%" FMT_blkcnt_t " nblock=%d\n", recno,
5267 #ifdef DEBUG      DEBUG("backtape() called, recno=%" FMT_blkcnt_t " nblock=%d\n", recno,
5268             nblock);
5269 #endif
5270     /*
5271      * Backup to the position in the archive where the record
5272      * currently sitting in the tbuf buffer is situated.
5273     */
5274     if (NotTape) {
5275         /*
5276          * For non-tape devices, this means lseeking to the
5277          * correct position. The absolute location tapepos-recno
5278          * should be the beginning of the current record.
5279         */
5280         if (lseek(mt, (off_t)(TBLOCK*(tapepos-recno)), SEEK_SET) ==
5281             (off_t)-1) {
5282             (void) fprintf(stderr,
5283                         gettext("tar: lseek to end of archive failed\n"));
5284             done(4);
5285         } else {
5286             /*
5287              * For tape devices, we backup over the most recently
5288              * read record.
5289             */
5290             mtcmd.mt_op = MTBSR;
5291             mtcmd.mt_count = 1;
5292
5293

```

```

5295     if (ioctl(mt, MTIOCTOP, &mtcmd) < 0) {
5296         (void) fprintf(stderr,
5297                     gettext("tar: backspace over record failed\n"));
5298         done(4);
5299     }
5300 }
5302 /*
5303  * Decrement the tape and tbuf buffer indices to prepare for the
5304  * coming write to overwrite the soft EOF record.
5305 */
5307     recno--;
5308     tapepos--;
5309 }

5312 /*
5313  * flushtape  write buffered block(s) onto tape
5314 *
5315  * recno points to next free block in tbuf. If nonzero, a write is done.
5316  * Care is taken to write in multiples of SYS_BLOCK when device is
5317  * non-magtape in case raw i/o is used.
5318 *
5319  * NOTE: this is called by writetape() to do the actual writing
5320 */
5322 static void
5323 flushtape(void)
5324 {
5325     dlog("flushtape() called, recno=%" FMT_blkcnt_t "\n", recno);
5281 #ifdef DEBUG      DEBUG("flushtape() called, recno=%" FMT_blkcnt_t "\n", recno, 0);
5282 #endif
5326     if (recno > 0) { /* anything buffered? */
5327         if (NotTape) {
5328             #if SYS_BLOCK > TBLOCK
5329                 int i;
5330
5331                 /*
5332                  * an odd-block write can only happen when
5333                  * we are at the end of a volume that is not a tape.
5334                  * Here we round recno up to an even SYS_BLOCK
5335                  * boundary.
5336                 */
5337                 if ((i = recno % (SYS_BLOCK / TBLOCK)) != 0) {
5338                     dlog("flushtape() %d rounding blocks\n", i);
5296 #ifdef DEBUG      DEBUG("flushtape() %d rounding blocks\n", i, 0);
5297 #endif
5339                     recno += i; /* round up to even SYS_BLOCK */
5340                 }
5341             #endif
5342             if (recno > nblock)
5343                 recno = nblock;
5344         }
5345         dlog("writing out %" FMT_blkcnt_t " blocks of %" FMT_blkcnt_t
5306             DEBUG("writing out %" FMT_blkcnt_t " blocks of %" FMT_blkcnt_t
5346             " bytes\n", (blkcnt_t)(NotTape ? recno : nblock),
5347             (blkcnt_t)(NotTape ? recno : nblock) * TBLOCK);
5309 #endif
5348         if (write(mt, tbuf,
5349             (size_t)(NotTape ? recno : nblock) * TBLOCK) < 0) {
5350             (void) fprintf(stderr, gettext(
5351                 "tar: tape write error\n"));
5352

```

```

5352         done(2);
5353     }
5354     recno = 0;
5355 }
5356 }  

unchanged_portion_omitted_  

5423 /*
5424 * defset()
5425 *   - reads DEF_FILE for the set of default values specified.
5426 *   - initializes 'usefile', 'nblock', and 'blocklim', and 'NotTape'.
5427 *   - 'usefile' points to static data, so will be overwritten
5428 *     if this routine is called a second time.
5429 *   - the pattern specified by 'arch' must be followed by four
5430 *     blank-separated fields (1) device (2) blocking,
5431 *           (3) size(K), and (4) tape
5432 *     for example: archive0=/dev/fd 1 400 n
5433 */  

5435 static int
5436 defset(char *arch)
5437 {
5438     char *bp;
5439
5440     if (defopen(DEF_FILE) != 0)
5441         return (FALSE);
5442     if (fcntl(DC_SETFLAGS, (DC_STD & ~(DC_CASE))) == -1) {
5443         (void) fprintf(stderr, gettext(
5444             "tar: error setting parameters for %s.\n"), DEF_FILE);
5445         return (FALSE); /* & following ones too */
5446     }
5447     if ((bp = fread(arch)) == NULL) {
5448         (void) fprintf(stderr, gettext(
5449             "tar: missing or invalid '%s' entry in %s.\n"),
5450             arch, DEF_FILE);
5451         return (FALSE);
5452     }
5453     if ((usefile = strtok(bp, " \t")) == NULL) {
5454         (void) fprintf(stderr, gettext(
5455             "tar: '%s' entry in %s is empty!\n"), arch, DEF_FILE);
5456         return (FALSE);
5457     }
5458     if ((bp = strtok(NULL, " \t")) == NULL) {
5459         (void) fprintf(stderr, gettext(
5460             "tar: block component missing in '%s' entry in %s.\n"),
5461             arch, DEF_FILE);
5462         return (FALSE);
5463     }
5464     nblock = bcheck(bp);
5465     if ((bp = strtok(NULL, " \t")) == NULL) {
5466         (void) fprintf(stderr, gettext(
5467             "tar: size component missing in '%s' entry in %s.\n"),
5468             arch, DEF_FILE);
5469         return (FALSE);
5470     }
5471     blocklim = kcheck(bp);
5472     if ((bp = strtok(NULL, " \t")) != NULL)
5473         NotTape = (*bp == 'n' || *bp == 'N');
5474     else
5475         NotTape = (blocklim != 0);
5476     (void) defopen(NULL);
5477     dlog("defset: archive='%"S"'; usefile='%"S"\n", arch, usefile);
5478     dlog("defset: nblock='%"d"'; blocklim='%"FMT_blkcnt_t "'\n",
5479 #ifdef DEBUG
5480     DEBUG("defset: archive='%"S"'; usefile='%"S"\n", arch, usefile),

```

```

5441     DEBUG("defset: nblock='%"d'; blocklim='%" FMT_blkcnt_t "'\n",
5442           nblock, blocklim);
5443     dlog("defset: not tape = %d\n", NotTape);
5444     DEBUG("defset: not tape = %d\n", NotTape, 0);
5445 }
5446 #endif
5447 return (TRUE);
5448 }  

unchanged_portion_omitted_  

5706 static int
5707 wantit(char *argv[], char **namep, char **dirp, char **component,
5708          attr_data_t **attrinfo)
5709 {
5710     char **cp;
5711     int gotit; /* true if we've found a match */
5712     int ret;
5713
5714 top:
5715     if (xhdr_flg & _X_XHDR) {
5716         xhdr_flg = 0;
5717     }
5718     getdir();
5719     if (Xhdrflflag > 0) {
5720         ret = get_xdata();
5721         if (ret != 0) /* Xhdr items and regular header */
5722             setbytes_to_skip(&stbuf, ret);
5723             passtape();
5724             return (0); /* Error--don't want to extract */
5725     }
5726 }
5727
5728 /*
5729 * If typeflag is not 'A' and xhdr_flg is set, then processing
5730 * of ancillary file is either over or ancillary file
5731 * processing is not required, load info from Xtarhdr and set
5732 * _X_XHDR bit in xhdr_flg.
5733 */
5734 if ((dblock.dbuf.typeflag != 'A') && (xhdr_flg != 0)) {
5735     load_info_from_xtarhdr(xhdr_flg, &Xtarhdr);
5736     xhdr_flg |= _X_XHDR;
5737 }
5738
5739 #if defined(O_XATTR)
5740     if (dblock.dbuf.typeflag == _XATTR_HDRTYPE && xatrbadhead == 0) {
5741         /*
5742          * Always needs to read the extended header. If atflag, saflag,
5743          * or tflag isn't set, then we'll have the correct info for
5744          * passtape() later.
5745         */
5746         (void) read_xattr_hdr(attrinfo);
5747         goto top;
5748     }
5749 */
5750     * Now that we've read the extended header, call passtape()
5751     * if we don't want to restore attributes or system attributes.
5752     * Don't restore the attribute if we are extracting
5753     * a file from an archive (as opposed to doing a table of
5754     * contents) and any of the following are true:
5755     * 1. neither -@ or -/ was specified.
5756     * 2. -@ was specified, -/ wasn't specified, and we're
5757     *    processing a hidden attribute directory of an attribute
5758     *    or we're processing a read-write system attribute file.
5759     * 3. -@ wasn't specified, -/ was specified, and the file
5760     *    we're processing is not a read-write system attribute file,
5761     *    or we're processing the hidden attribute directory of an
5762     *    attribute.

```

```

5763     *
5764     * We always process the attributes if we're just generating
5765     * generating a table of contents, or if both -@ and -/ were
5766     * specified.
5767     */
5768     if (xattrp != NULL) {
5769         attr_data_t *ainfo = *attrinfo;
5770
5771         if (!tfflag &&
5772             ((!atflag && !saflag) ||
5773              (atflag && !saflag && ((ainfo->attr_parent != NULL) ||
5774                ainfo->attr_rw_sysattr)) ||
5775              (!atflag && saflag && ((ainfo->attr_parent != NULL) ||
5776                ainfo->attr_rw_sysattr))) {
5777             passtape();
5778             return (0);
5779         }
5780     }
5781 #endif
5782
5783     /* sets *namep to point at the proper name */
5784     if (check_prefix(namep, dirp, component) != 0) {
5785         passtape();
5786         return (0);
5787     }
5788
5789     if (endtape()) {
5790         if (Bflag) {
5791             ssize_t sz;
5792             size_t extra_blocks = 0;
5793
5794             /*
5795             * Logically at EOT - consume any extra blocks
5796             * so that write to our stdin won't fail and
5797             * emit an error message; otherwise something
5798             * like "dd if=foo.tar | (cd bar; tar xvf -)"
5799             * will produce a bogus error message from "dd".
5800             */
5801
5802             while ((sz = read(mt, tbuf, TBLOCK*nblock)) > 0) {
5803                 extra_blocks += sz;
5804                 while (read(mt, tbuf, TBLOCK*nblock) > 0) {
5805                     /* empty body */
5806                     dlog("wantit(): %d bytes of extra blocks\n",
5807                         extra_blocks);
5808                 }
5809                 dlog("wantit(): at end of tape.\n");
5810                 return (-1);
5811             }
5812             gotit = 0;
5813
5814             if ((Iflag && is_in_table(include_tbl, *namep)) ||
5815                 (! Iflag && *argv == NULL)) {
5816                 gotit = 1;
5817             } else {
5818                 for (cp = argv; *cp; cp++) {
5819                     if (is_prefix(*cp, *namep)) {
5820                         gotit = 1;
5821                         break;
5822                     }
5823                 }
5824             }
5825             if (! gotit) {

```

```

5826                 passtape();
5827                 return (0);
5828             }
5829
5830             if (Xflag && is_in_table(exclude_tbl, *namep)) {
5831                 if (vflag) {
5832                     (void) fprintf(stderr, gettext("%s excluded\n"),
5833                         *namep);
5834                 }
5835                 passtape();
5836                 return (0);
5837             }
5838         }
5839     }
5840     return (1);
5841 }
5842 unchanged_portion_omitted
5843 #endif /* O_XATTR */
5844
5845 static int
5846 put_link(char *name, char *longname, char *component, char *longattrname,
5847           char *prefix, int filetype, char type)
5848 {
5849
5850     if (stbuf.st_nlink > 1) {
5851         struct linkbuf *lp;
5852         int found = 0;
5853
5854         for (lp = ihead; lp != NULL; lp = lp->nextp)
5855             if (lp->inum == stbuf.st_ino &&
5856                 lp->devnum == stbuf.st_dev) {
5857                 found++;
5858                 break;
5859             }
5860         if (found) {
5861             if (filetype == XATTR_FILE)
5862                 if (put_xattr_hdr(longname, component,
5863                                   longattrname, prefix, type, filetype, lp))
5864                     goto out;
5865         }
5866     }
5867     stbuf.st_size = (off_t)0;
5868     if (filetype != XATTR_FILE) {
5869         tomodes(&stbuf);
5870         if (chk_path_build(name, longname, lp->pathname,
5871                           prefix, type, filetype) > 0) {
5872             goto out;
5873         }
5874     }
5875
5876     if (mulvol && tapepos + 1 >= blocklim)
5877         newvol();
5878     (void) writetbuf((char *)&dblock, 1);
5879     /*
5880      * write_ancillary() is not needed here.
5881      * The first link is handled in the following
5882      * else statement. No need to process ACLs
5883      * for other hard links since they are the
5884      * same file.
5885     */
5886
5887     if (vflag) {
5888         if (NotTape)
5889             #ifdef DEBUG
5890                 dlog("seek = %" FMT_blkcnt_t
5891                      "K\n", K(tapepos));

```

```
8021                     DEBUG("seek = %" FMT_blkcnt_t
8022                     "K\t", K(tapepos), 0);
8023 #endif
8024     if (filetype == XATTR_FILE) {
8025         (void) fprintf(vfile, gettext(
8026             "a %s attribute %s link to "
8027             "%s attribute %s\n"),
8028             name, component, name,
8029             lp->attrname);
8030     } else {
8031         (void) fprintf(vfile, gettext(
8032             "a %s link to %s\n"),
8033             longname, lp->pathname);
8034     }
8035     lp->count--;
8036 }
8037 return (0);
8038 } else {
8039     lp = (struct linkbuf *)getmem(sizeof (*lp));
8040     if (lp != (struct linkbuf *)NULL) {
8041         lp->nextp = ihead;
8042         ihead = lp;
8043         lp->inum = stbuf.st_ino;
8044         lp->devnum = stbuf.st_dev;
8045         lp->count = stbuf.st_nlink - 1;
8046         if (filetype == XATTR_FILE) {
8047             (void) strcpy(lp->pathname, longname);
8048             (void) strcpy(lp->attrname,
8049                 component);
8050         } else {
8051             (void) strcpy(lp->pathname, longname);
8052             (void) strcpy(lp->attrname, "");
8053         }
8054     }
8055 }
8056 }
8057
8058 out:
8059     return (1);
8060 }
```

unchanged_portion_omitted_