```
**********************************************************
   28755 Thu Jan 17 14:27:12 2019
new/usr/src/cmd/audio/audioplay/audioplay.c
10101 audio tools need smatch fixes
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2018, Joyent, Inc.
 */

/* Command-line audio play utility */

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <signal.h>
#include <locale.h>
#include <limits.h>     /* All occurances of INT_MAX used to be ~0  (by MCA) */
#include <unistd.h>
#include <stropts.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <netinet/in.h>

#include <libaudio.h>
#include <audio_device.h>
#include <audio_encode.h>

/* localization stuff */
#define MGET(s)         (char *)gettext(s)

#if !defined(TEXT_DOMAIN)       /* Should be defined by cc -D */
#define TEXT_DOMAIN "SYS_TEST"  /* Use this only if it weren't */
#endif
```



```
#define Error           (void) fprintf


/* Local variables */
static char *prog;

static char prog_opts[] =       "VEiv:d:?";     /* getopt() flags */

static char                     *Stdin;

#define MAX_GAIN                (100)           /* maximum gain */

/*
 * This defines the tolerable sample rate error as a ratio between the
 * sample rates of the audio data and the audio device.
 */
#define SAMPLE_RATE_THRESHOLD   (.01)

#define         BUFFER_LEN      10      /* seconds - for file i/o */
#define         ADPCM_SIZE      (1000*8) /* adpcm conversion output buf size */
#define         SWAP_SIZE       (8192)
                        /* swap bytes conversion output buf size */

static unsigned         Volume = INT_MAX;       /* output volume */
static double           Savevol;                /* saved volume level */

static int              Verbose = FALSE;        /* verbose messages */
static int              Immediate = FALSE;
                        /* don't hang waiting for device */
static int              Errdetect = FALSE;      /* don't worry about underrun */
static char             *Audio_dev = "/dev/audio";

static int NetEndian = TRUE;            /* endian nature of the machine */

static int              Audio_fd = -1;
                        /* file descriptor for audio device */
static int              Audio_ctlfd = -1;
                        /* file descriptor for control device */
static Audio_hdr        Save_hdr;
                        /* saved audio header for device */
static Audio_hdr        Dev_hdr;                /* audio header for device */
static char             *Ifile;                 /* current filename */
static Audio_hdr        File_hdr;               /* audio header for file */
static unsigned         Decode = AUDIO_ENCODING_NONE;
                        /* decode type, if any */

static unsigned char    *buf = NULL;            /* dynamically alloc'd */
static unsigned         bufsiz = 0;             /* size of output buffer */
static unsigned char    adpcm_buf[ADPCM_SIZE + 32];
                        /* for adpcm conversion */
static unsigned char    swap_buf[SWAP_SIZE + 32];
                        /* for byte swap conversion */
static unsigned char    *inbuf;
                        /* current input buffer pointer */
static unsigned         insiz;                  /* current input buffer size */

/*
 * The decode_g72x() function is capable of decoding only one channel
 * at a time and so multichannel data must be decomposed (using demux()
 * function below ) into its constituent channels and each passed
 * separately to the decode_g72x() function. Encoded input channels are
 * stored in **in_ch_data and decoded output channels in **out_ch_data.
 * Once each channel has been decoded they are recombined (see mux()
 * function below) before being written to the audio device. For each
 * channel and adpcm state structure is created.
 */
```

```
129 /* adpcm state structures */
130 static struct audio_g72x_state *adpcm_state = NULL;
131 static unsigned char    **in_ch_data = NULL;    /* input channels */
132 static unsigned char    **out_ch_data = NULL;   /* output channels */
133 static int              out_ch_size;            /* output channel size */

135 static char             *Audio_path = NULL;
136                         /* path to search for audio files */

138 /* Global variables */
139 extern int      getopt(int, char *const *, const char *);
140 extern int      optind;
141 extern char     *optarg;

143 /* Local functions  */
144 static void usage(void);
145 static void sigint(int sig);
146 static void open_audio(void);
147 static int path_open(char *fname, int flags, mode_t mode, char *path);
148 static int parse_unsigned(char *str, unsigned *dst, char *flag);
149 static int reconfig(void);
150 static void initmux(int unitsz, int unitsp);
151 static void demux(int unitsz, int cnt);
152 static void mux(char *);
153 static void freemux(void);


156 static void
157 usage(void)
158 {
159         Error(stderr, MGET("Play an audio file -- usage:\n"
160             "\t%s [-iV] [-v vol] [-d dev] [file ...]\n"
161             "where:\n"
162             "\t-i\tDon't hang if audio device is busy\n"
163             "\t-V\tPrint verbose warning messages\n"
164             "\t-v\tSet output volume (0 - %d)\n"
165             "\t-d\tSpecify audio device (default: /dev/audio)\n"
166             "\tfile\tList of files to play\n"
167             "\t\tIf no files specified, read stdin\n"),
168             prog, MAX_GAIN);
169         exit(1);
170 }
```

*unchanged_portion_omitted_*

```
250 /* Play a list of audio files. */
251 int
252 main(int argc, char **argv) {
253         int             errorStatus = 0;
254         int             i;
255         int             c;
256         int             cnt;
257         int             file_type;
258         int             rem;
259         int             outsiz;
260         int             tsize;
261         int             len;
262         int             err;
263         int             ifd;
264         int             stdinseen;
265         int             regular;
266         int             swapBytes;
267         int             frame;
268         char            *outbuf;
269         caddr_t         mapaddr;
270         struct stat     st;
```



```
271         char            *cp;
272         char            ctldev[MAXPATHLEN];

274         (void) setlocale(LC_ALL, "");
275         (void) textdomain(TEXT_DOMAIN);

277         /* Get the program name */
278         prog = strrchr(argv[0], '/');
279         if (prog == NULL)
280                 prog = argv[0];
281         else
282                 prog++;
283         Stdin = MGET("(stdin)");

285         /* Check AUDIODEV environment for audio device name */
286         if (cp = getenv("AUDIODEV")) {
287                 Audio_dev = cp;
288         }

290         /* Parse the command line arguments */
291         err = 0;
292         while ((i = getopt(argc, argv, prog_opts)) != EOF) {
293                 switch (i) {
294                         case 'v':
295                                 if (parse_unsigned(optarg, &Volume, "-v")) {
296                                         err++;
297                                 } else if (Volume > MAX_GAIN) {
298                                         Error(stderr, MGET("%s: invalid value "
299                                             "for -v\n"), prog);
300                                         err++;
301                                 }
302                                 break;
303                         case 'd':
304                                 Audio_dev = optarg;
305                                 break;
306                         case 'V':
307                                 Verbose = TRUE;
308                                 break;
309                         case 'E':
310                                 Errdetect = TRUE;
311                                 break;
312                         case 'i':
313                                 Immediate = TRUE;
314                                 break;
315                         case '?':
316                                 usage();
317                 /*NOTREACHED*/
318                 }
319         }
320         if (err > 0)
321                 exit(1);

323         argc -= optind;         /* update arg pointers */
324         argv += optind;

326         /* Validate and open the audio device */
327         err = stat(Audio_dev, &st);
328         if (err < 0) {
329                 Error(stderr, MGET("%s: cannot stat "), prog);
330                 perror(Audio_dev);
331                 exit(1);
332         }
333         if (!S_ISCHR(st.st_mode)) {
334                 Error(stderr, MGET("%s: %s is not an audio device\n"), prog,
335                     Audio_dev);
336                 exit(1);
```

```
337         }
339         /* This should probably use audio_cntl instead of open_audio */
340         if ((argc <= 0) && isatty(fileno(stdin))) {
341                 Error(stderr, MGET("%s: No files and stdin is a tty.\n"), prog);
342                 exit(1);
343         }
345         /* Check on the -i status now. */
346         Audio_fd = open(Audio_dev, O_WRONLY | O_NONBLOCK);
347         if ((Audio_fd < 0) && (errno == EBUSY)) {
348                 if (Immediate) {
349                         Error(stderr, MGET("%s: %s is busy\n"), prog,
350                             Audio_dev);
351                         exit(1);
352                 }
353         }
354         (void) close(Audio_fd);
355         Audio_fd = -1;
357         /* Try to open the control device and save the current format */
358         (void) snprintf(ctldev, sizeof (ctldev), "%sctl", Audio_dev);
359         Audio_ctlfd = open(ctldev, O_RDWR);
360         if (Audio_ctlfd >= 0) {
361                 /*
362                  * wait for the device to become available then get the
363                  * controls. We want to save the format that is left when the
364                  * device is in a quiescent state. So wait until then.
365                  */
366                 Audio_fd = open(Audio_dev, O_WRONLY);
367                 (void) close(Audio_fd);
368                 Audio_fd = -1;
369                 if (audio_get_play_config(Audio_ctlfd, &Save_hdr)
370                     != AUDIO_SUCCESS) {
371                         (void) close(Audio_ctlfd);
372                         Audio_ctlfd = -1;
373                 }
374         }
376         /* store AUDIOPATH so we don't keep doing getenv() */
377         Audio_path = getenv("AUDIOPATH");
379         /* Set up SIGINT handler to flush output */
380         (void) signal(SIGINT, sigint);
382         /* Set the endian nature of the machine. */
383         if ((ulong_t)1 != htonl((ulong_t)1)) {
384                 NetEndian = FALSE;
385         }
387         /* If no filenames, read stdin */
388         stdinseen = FALSE;
389         if (argc <= 0) {
390                 Ifile = Stdin;
391         } else {
392                 Ifile = *argv++;
393                 argc--;
394         }
396         /* Loop through all filenames */
397         do {
398                 /* Interpret "-" filename to mean stdin */
399                 if (strcmp(Ifile, "-") == 0)
400                         Ifile = Stdin;
402                 if (Ifile == Stdin) {
```



```
403                         if (stdinseen) {
404                                 Error(stderr,
405                                     MGET("%s: stdin already processed\n"),
406                                     prog);
407                                 goto nextfile;
408                         }
409                         stdinseen = TRUE;
410                         ifd = fileno(stdin);
411                 } else {
412                         if ((ifd = path_open(Ifile, O_RDONLY, 0, Audio_path))
413                             < 0) {
414                                 Error(stderr, MGET("%s: cannot open "), prog);
415                                 perror(Ifile);
416                                 errorStatus++;
417                                 goto nextfile;
418                         }
419                 }
421                 /* Check to make sure this is an audio file */
422                 err = audio_read_filehdr(ifd, &File_hdr, &file_type,
423                     (char *)NULL, 0);
424                 if (err != AUDIO_SUCCESS) {
425                         Error(stderr,
426                             MGET("%s: %s is not a valid audio file\n"),
427                             prog, Ifile);
428                         errorStatus++;
429                         goto closeinput;
430                 }
432                 /* If G.72X adpcm, set flags for conversion */
433                 if ((File_hdr.encoding == AUDIO_ENCODING_G721) &&
434                     (File_hdr.samples_per_unit == 2) &&
435                     (File_hdr.bytes_per_unit == 1)) {
436                         Decode = AUDIO_ENCODING_G721;
437                         File_hdr.encoding = AUDIO_ENCODING_ULAW;
438                         File_hdr.samples_per_unit = 1;
439                         File_hdr.bytes_per_unit = 1;
440                         adpcm_state = (struct audio_g72x_state *)malloc
441                             (sizeof (*adpcm_state) * File_hdr.channels);
442                         for (i = 0; i < File_hdr.channels; i++) {
443                                 g721_init_state(&adpcm_state[i]);
444                         }
445                 } else if ((File_hdr.encoding == AUDIO_ENCODING_G723) &&
446                     (File_hdr.samples_per_unit == 8) &&
447                     (File_hdr.bytes_per_unit == 3)) {
448                         Decode = AUDIO_ENCODING_G723;
449                         File_hdr.encoding = AUDIO_ENCODING_ULAW;
450                         File_hdr.samples_per_unit = 1;
451                         File_hdr.bytes_per_unit = 1;
452                         adpcm_state = (struct audio_g72x_state *)malloc
453                             (sizeof (*adpcm_state) * File_hdr.channels);
454                         for (i = 0; i < File_hdr.channels; i++) {
455                                 g723_init_state(&adpcm_state[i]);
456                         }
457                 } else {
458                         Decode = AUDIO_ENCODING_NONE;
459                 }
461                 /* Check the device configuration */
462                 open_audio();
463                 if (audio_cmp_hdr(&Dev_hdr, &File_hdr) != 0) {
464                         /*
465                          * The device does not match the input file.
466                          * Wait for any old output to drain, then attempt
467                          * to reconfigure the audio device to match the
468                          * input data.
```

```
469                          */
470                         if (audio_drain(Audio_fd, FALSE) != AUDIO_SUCCESS) {
471                                 /* Flush any remaining audio */
472                                 (void) ioctl(Audio_fd, I_FLUSH, FLUSHW);
474                                 Error(stderr, MGET("%s: "), prog);
475                                 perror(MGET("AUDIO_DRAIN error"));
476                                 exit(1);
477                         }
479                         /* Flush any remaining audio */
480                         (void) ioctl(Audio_fd, I_FLUSH, FLUSHW);
482                         if (!reconfig()) {
483                                 errorStatus++;
484                                 goto closeinput;
485                         }
486                 }
489                 /* try to do the mmaping - for regular files only ... */
490                 err = fstat(ifd, &st);
491                 if (err < 0) {
492                         Error(stderr, MGET("%s: cannot stat "), prog);
493                         perror(Ifile);
494                         exit(1);
495                 }
496                 regular = (S_ISREG(st.st_mode));
499                 /* If regular file, map it.  Else, allocate a buffer */
500                 mapaddr = 0;
502                 /*
503                  * This should compare to MAP_FAILED not -1, can't
504                  * find MAP_FAILED
505                  */
506                 if (regular && ((mapaddr = mmap(0, st.st_size, PROT_READ,
507                     MAP_SHARED, ifd, 0)) != MAP_FAILED)) {
509                         (void) madvise(mapaddr, st.st_size, MADV_SEQUENTIAL);
511                         /* Skip the file header and set the proper size */
512                         cnt = lseek(ifd, 0, SEEK_CUR);
513                         if (cnt < 0) {
514                             perror("lseek");
515                             exit(1);
516                         }
517                         inbuf = (unsigned char *) mapaddr + cnt;
518                         len = cnt = st.st_size - cnt;
519                 } else {                /* Not a regular file, or map failed */
521                         /* mark is so. */
522                         mapaddr = 0;
524                         /* Allocate buffer to hold 10 seconds of data */
525                         cnt = BUFFER_LEN * File_hdr.sample_rate *
526                             File_hdr.bytes_per_unit * File_hdr.channels;
527                         if (bufsiz != cnt) {
528                                 if (buf != NULL) {
529                                         (void) free(buf);
530                                 }
531                                 buf = (unsigned char *) malloc(cnt);
532                                 if (buf == NULL) {
533                                         Error(stderr,
534                                             MGET("%s: couldn't allocate %dK "
```



```
535                                             "buf\n"), prog, bufsiz / 1000);
536                                         exit(1);
537                                 }
538                                 inbuf = buf;
539                                 bufsiz = cnt;
540                         }
541                 }
543                 /* Set buffer sizes and pointers for conversion, if any */
544                 switch (Decode) {
545                 default:
546                 case AUDIO_ENCODING_NONE:
547                         insiz = bufsiz;
548                         outbuf = (char *)buf;
549                         break;
550                 case AUDIO_ENCODING_G721:
551                         insiz = ADPCM_SIZE / 2;
552                         outbuf = (char *)adpcm_buf;
553                         initmux(1, 2);
554                         break;
555                 case AUDIO_ENCODING_G723:
556                         insiz = (ADPCM_SIZE * 3) / 8;
557                         outbuf = (char *)adpcm_buf;
558                         initmux(3, 8);
559                         break;
560                 }
562                 /*
563                  * 8-bit audio isn't a problem, however 16-bit audio is.
564                  * If the file is an endian that is different from the machine
565                  * then the bytes will need to be swapped.
566                  *
567                  * Note: Because the G.72X conversions produce 8bit output,
568                  * they don't require a byte swap before display and so
569                  * this scheme works just fine. If a conversion is added
570                  * that produces a 16 bit result and therefore requires
571                  * byte swapping before output, then a mechanism
572                  * for chaining the two conversions will have to be built.
573                  *
574                  * Note: The following if() could be simplified, but then
575                  * it gets to be very hard to read. So it's left as is.
576                  */
578                 if (File_hdr.bytes_per_unit == 2 &&
579                     ((!NetEndian && file_type == FILE_AIFF) ||
580                     (!NetEndian && file_type == FILE_AU) ||
581                     (NetEndian && file_type == FILE_WAV))) {
582                         swapBytes = TRUE;
583                 } else {
584                         swapBytes = FALSE;
585                 }
587                 if (swapBytes) {
588                         /* Read in interal number of sample frames. */
589                         frame = File_hdr.bytes_per_unit * File_hdr.channels;
590                         insiz = (SWAP_SIZE / frame) * frame;
591                         /* make the output buffer  the swap buffer. */
592                         outbuf = (char *)swap_buf;
593                 }
595                 /*
596                  * At this point, we're all ready to copy the data.
597                  */
598                 if (mapaddr == 0) { /* Not mmapped, do it a buffer at a time. */
599                         inbuf = buf;
600                         frame = File_hdr.bytes_per_unit * File_hdr.channels;
```

```
601                         rem = 0;
602                         while ((cnt = read(ifd, inbuf+rem, insiz-rem)) >= 0) {
603                                 /*
604                                  * We need to ensure only an integral number of
605                                  * samples is ever written to the audio device.
606                                  */
607                                 cnt = cnt + rem;
608                                 rem = cnt % frame;
609                                 cnt = cnt - rem;

611                                 /*
612                                  * If decoding adpcm, or swapping bytes do it
613                                  * now.
614                                  *
615                                  * We treat the swapping like a separate
616                                  * encoding here because the G.72X encodings
617                                  * decode to single byte output samples. If
618                                  * another encoding is added and it produces
619                                  * multi-byte output samples this will have to
620                                  * be changed.
621                                  */
622                                 if (Decode == AUDIO_ENCODING_G721) {
623                                     outsiz = 0;
624                                     demux(1, cnt / File_hdr.channels);
625                                     for (c = 0; c < File_hdr.channels; c++) {
626                                         err = g721_decode(in_ch_data[c],
627                                             cnt / File_hdr.channels,
628                                             &File_hdr,
629                                             (void*)out_ch_data[c],
630                                             &tsize,
631                                             &adpcm_state[c]);
632                                         outsiz = outsiz + tsize;
633                                         if (err != AUDIO_SUCCESS) {
634                                             Error(stderr, MGET(
635                                                 "%s: error decoding g721\n"),
636                                                 prog);
637                                                 errorStatus++;
638                                                 break;
639                                         }
640                                     }
641                                     mux(outbuf);
642                                     cnt = outsiz;
643                                 } else if (Decode == AUDIO_ENCODING_G723) {
644                                     outsiz = 0;
645                                     demux(3, cnt / File_hdr.channels);
646                                     for (c = 0; c < File_hdr.channels; c++) {
647                                         err = g723_decode(in_ch_data[c],
648                                             cnt / File_hdr.channels,
649                                             &File_hdr,
650                                             (void*)out_ch_data[c],
651                                             &tsize,
652                                             &adpcm_state[c]);
653                                         outsiz = outsiz + tsize;
654                                         if (err != AUDIO_SUCCESS) {
655                                             Error(stderr, MGET(
656                                                 "%s: error decoding g723\n"),
657                                                 prog);
658                                             errorStatus++;
659                                             break;
660                                         }
661                                     }
662                                     mux(outbuf);
663                                     cnt = outsiz;
664                                 } else if (swapBytes) {
665                                         swab((char *)inbuf, outbuf, cnt);
666                                 }
```



```
668                                 /* If input EOF, write an eof marker */
669                                 err = write(Audio_fd, outbuf, cnt);

671                                 if (err < 0) {
672                                         perror("write");
673                                         errorStatus++;
674                                         break;
675                                 } else if (err != cnt) {
676                                         Error(stderr,
677                                             MGET("%s: output error: "), prog);
678                                         perror("");
679                                         errorStatus++;
680                                         break;
681                                 }
682                                 if (cnt == 0) {
683                                         break;
684                                 }
685                                 /* Move remainder to the front of the buffer */
686                                 if (rem != 0) {
687                                         (void *)memcpy(inbuf, inbuf + cnt, rem);
688                                 }

690                         }
691                         if (cnt < 0) {
692                                 Error(stderr, MGET("%s: error reading "), prog);
693                                 perror(Ifile);
694                                 errorStatus++;
695                         }
696                 } else {        /* We're mmaped */
697                         if ((Decode != AUDIO_ENCODING_NONE) || swapBytes) {

699                                 /* Transform data if we have to. */
700                                 for (i = 0; i <= len; i += cnt) {
701                                         cnt = insiz;
702                                         if ((i + cnt) > len) {
703                                                 cnt = len - i;
704                                         }
705                                         if (Decode == AUDIO_ENCODING_G721) {
706                                             outsiz = 0;
707                                             demux(1, cnt / File_hdr.channels);
708                                             for (c = 0; c < File_hdr.channels;
709                                                 c++) {
710                                                 err = g721_decode(
711                                                     in_ch_data[c],
712                                                     cnt / File_hdr.channels,
713                                                     &File_hdr,
714                                                     (void*)out_ch_data[c],
715                                                     &tsize,
716                                                     &adpcm_state[c]);
717                                                 outsiz = outsiz + tsize;
718                                                 if (err != AUDIO_SUCCESS) {
719                                                     Error(stderr, MGET(
720                                                         "%s: error decoding "
721                                                         "g721\n"), prog);
722                                                     errorStatus++;
723                                                     break;
724                                                 }
725                                             }
726                                             mux(outbuf);
727                                         } else if
728                                             (Decode == AUDIO_ENCODING_G723) {
729                                                 outsiz = 0;
730                                                 demux(3,
731                                                     cnt / File_hdr.channels);
732                                                 for (c = 0;
```

```
                                                c < File_hdr.channels;
                                                c++) {
                                            err = g723_decode(
                                                in_ch_data[c],
                                                cnt /
                                                    File_hdr.channels,
                                                &File_hdr,
                                                (void*)out_ch_data[c],
                                                &tsize,
                                                &adpcm_state[c]);
                                            outsiz = outsiz + tsize;
                                            if (err != AUDIO_SUCCESS) {
                                                Error(stderr, MGET(
                                                    "%s: error "
                                                    "decoding g723\n"),
                                                    prog);
                                                errorStatus++;
                                                break;
                                            }
                                        }
                                        mux(outbuf);
                                    } else if (swapBytes) {
                                        swab((char *)inbuf, outbuf,
                                            cnt);
                                        outsiz = cnt;
                                    }
                                    inbuf += cnt;

                                    /* If input EOF, write an eof marker */
                                    err = write(Audio_fd, (char *)outbuf,
                                        outsiz);
                                    if (err < 0) {
                                        perror("write");
                                        errorStatus++;
                                    } else if (outsiz == 0) {
                                        break;
                                    }

                            }
                        } else {
                            /* write the whole thing at once!  */
                            err = write(Audio_fd, inbuf, len);
                            if (err < 0) {
                                perror("write");
                                errorStatus++;
                            }
                            if (err != len) {
                                Error(stderr,
                                    MGET("%s: output error: "), prog);
                                perror("");
                                errorStatus++;
                            }
                            err = write(Audio_fd, inbuf, 0);
                            if (err < 0) {
                                perror("write");
                                errorStatus++;
                            }
                        }
                }

                /* Free memory if decoding ADPCM */
                switch (Decode) {
                case AUDIO_ENCODING_G721:
                case AUDIO_ENCODING_G723:
                        freemux();
                        break;
```



```
                default:
                        break;
                }

closeinput:;
                if (mapaddr != 0)
                        (void) munmap(mapaddr, st.st_size);
                (void) close(ifd);              /* close input file */
                if (Errdetect) {
                        cnt = 0;
                        (void) audio_set_play_error(Audio_fd,
                            (unsigned int *)&cnt);
                        audio_set_play_error(Audio_fd, (unsigned int *)&cnt);
                        if (cnt) {
                                Error(stderr,
                                    MGET("%s: output underflow in %s\n"),
                                    Ifile, prog);
                                errorStatus++;
                        }
                }
nextfile:;
        } while ((argc > 0) && (argc--, (Ifile = *argv++) != NULL));

        /*
         * Though drain is implicit on close(), it's performed here
         * to ensure that the volume is reset after all output is complete.
         */
        (void) audio_drain(Audio_fd, FALSE);

        /* Flush any remaining audio */
        (void) ioctl(Audio_fd, I_FLUSH, FLUSHW);

        if (Volume != INT_MAX)
                (void) audio_set_play_gain(Audio_fd, &Savevol);
        if ((Audio_ctlfd >= 0) && (audio_cmp_hdr(&Save_hdr, &Dev_hdr) != 0)) {
                (void) audio_set_play_config(Audio_fd, &Save_hdr);
        }
        (void) close(Audio_fd);                 /* close output */
        return (errorStatus);
}
```

unchanged_portion_omitted_

```
**********************************************************
   20769 Thu Jan 17 14:27:12 2019
new/usr/src/cmd/audio/audiorecord/audiorecord.c
10101 audio tools need smatch fixes
**********************************************************
```

```
/*
 * CDDL HEADER START
 *
 * The contents of this file are subject to the terms of the
 * Common Development and Distribution License (the "License").
 * You may not use this file except in compliance with the License.
 *
 * You can obtain a copy of the license at usr/src/OPENSOLARIS.LICENSE
 * or http://www.opensolaris.org/os/licensing.
 * See the License for the specific language governing permissions
 * and limitations under the License.
 *
 * When distributing Covered Code, include this CDDL HEADER in each
 * file and include the License file at usr/src/OPENSOLARIS.LICENSE.
 * If applicable, add the following below this CDDL HEADER, with the
 * fields enclosed by brackets "[]" replaced with your own identifying
 * information: Portions Copyright [yyyy] [name of copyright owner]
 *
 * CDDL HEADER END
 */
/*
 * Copyright 2009 Sun Microsystems, Inc.  All rights reserved.
 * Use is subject to license terms.
 */

/*
 * Copyright (c) 2018, Joyent, Inc.
 */

/* Command-line audio record utility */

#include <stdio.h>
#include <libgen.h>
#include <errno.h>
#include <ctype.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <locale.h>
#include <fcntl.h>
#include <signal.h>
#include <limits.h>     /* All occurances of INT_MAX used to be ~0  (by MCA) */
#include <sys/types.h>
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <stropts.h>
#include <poll.h>
#include <sys/ioctl.h>
#include <netinet/in.h>

#include <libaudio.h>
#include <audio_device.h>

#define irint(d)        ((int)d)

/* localization stuff */
#define MGET(s)         (char *)gettext(s)
```



```
#if !defined(TEXT_DOMAIN)       /* Should be defined by cc -D */
#define TEXT_DOMAIN "SYS_TEST"  /* Use this only if it weren't */
#endif

#define Error           (void) fprintf

/* Local variables */
static char     *prog;
static char     prog_opts[] = "aft:v:d:i:e:s:c:T:?"; /* getopt() flags */
static char     *Stdout;

/* XXX - the input buffer size should depend on sample_rate */
#define AUDIO_BUFSIZ (1024 * 64)
static unsigned char    buf[AUDIO_BUFSIZ];
static char             swapBuf[AUDIO_BUFSIZ];  /* for byte swapping */


#define MAX_GAIN                (100)   /* maximum gain */

static char     *Info = NULL;           /* pointer to info data */
static unsigned Ilen = 0;               /* length of info data */
static unsigned Volume = INT_MAX;       /* record volume */
static double   Savevol;                /* saved  volume */
static unsigned Sample_rate = 0;
static unsigned Channels = 0;
static unsigned Precision = 0;          /* based on encoding */
static unsigned Encoding = 0;

static int      NetEndian = TRUE;       /* endian nature of the machines */

static int      Append = FALSE;         /* append to output file */
static int      Force = FALSE;          /* ignore rate differences on append */
static double   Time = -1.;             /* recording time */
static unsigned Limit = AUDIO_UNKNOWN_SIZE;     /* recording limit */
static char     *Audio_dev = "/dev/audio";

static int              Audio_fd = -1;
                        /* file descriptor for audio device */
static Audio_hdr        Dev_hdr;                /* audio header for device */
static Audio_hdr        Save_hdr;               /* saved audio device header */
static char             *Ofile;                 /* current filename */
static int              File_type = FILE_AU;    /* audio file type */
static int              File_type_set = FALSE;  /* file type specified as arg */
static Audio_hdr        File_hdr;               /* audio header for file */
static int              Cleanup = FALSE;        /* SIGINT sets this flag */
static unsigned         Size = 0;               /* Size of output file */
static unsigned         Oldsize = 0;
                        /* Size of input file, if append */

/* Global variables */
extern int getopt();
extern int optind;
extern char *optarg;

/* Local Functions */
static void usage(void);
static void sigint(int sig);
static int parse_unsigned(char *str, unsigned *dst, char *flag);
static int parse_sample_rate(char *s, unsigned *rate);


static void
usage(void)
{
        Error(stderr, MGET("Record an audio file -- usage:\n"
            "\t%s [-af] [-v vol]\n"
```

```
128             "\t%.*s [-c channels] [-s rate] [-e encoding]\n"
129             "\t%.*s [-t time] [-i info] [-d dev] [-T au|wav|aif[f]] [file]\n"
130             "where:\n"
131             "\t-a\tAppend to output file\n"
132             "\t-f\tIgnore sample rate differences on append\n"
133             "\t-v\tSet record volume (0 - %d)\n"
134             "\t-c\tSpecify number of channels to record\n"
135             "\t-s\tSpecify rate in samples per second\n"
136             "\t-e\tSpecify encoding (ulaw | alaw | [u]linear | linear8 )\n"
137             "\t-t\tSpecify record time (hh:mm:ss.dd)\n"
138             "\t-i\tSpecify a file header information string\n"
139             "\t-d\tSpecify audio device (default: /dev/audio)\n"
140             "\t-T\tSpecify the audio file type (default: au)\n"
141             "\tfile\tRecord to named file\n"
142             "\t\tIf no file specified, write to stdout\n"
143             "\t\tDefault audio encoding is ulaw, 8khz, mono\n"
144             "\t\tIf -t is not specified, record until ^C\n"),
145             prog,
146             strlen(prog), "                    ",
147             strlen(prog), "                    ",
148             MAX_GAIN);
149         exit(1);
150 }
```

unchanged_portion_omitted_

```
741 /*
742  * set the sample rate. assume anything is ok. check later on to make sure
743  * the sample rate is valid.
744  */
745 static int
746 parse_sample_rate(char *s, unsigned *rate)
747 {
748         char            *cp;
749         double          drate;

751         /*
752          * check if it's "cd" or "dat" or "voice". these also set
753          * the precision and encoding, etc.
754          */
755         if (strcasecmp(s, "dat") == 0) {
756                 drate = 48000.0;
757         } else if (strcasecmp(s, "cd") == 0) {
758                 drate = 44100.0;
759         } else if (strcasecmp(s, "voice") == 0) {
760                 drate = 8000.0;
761         } else {
762                 /* just do an atof */
763                 drate = atof(s);

765                 /*
766                  * if the first non-digit is a "k" multiply by 1000,
767                  * if it's an "h", leave it alone. anything else,
768                  * return an error.
769                  */

771                 /*
772                  * XXX bug alert: could have multiple "." in string
773                  * and mess things up.
774                  */
775                 for (cp = s; *cp && (isdigit(*cp) || (*cp == '.')); cp++)
776                         /* NOP */;
777                 if (*cp != NULL) {
778                         if ((*cp == 'k') || (*cp == 'K')) {
779                                 drate *= 1000.0;
780                         } else if ((*cp != 'h') && (*cp != 'H')) {
776                         } else if ((*cp != 'h') || (*cp != 'H')) {
```



```
781                                 /* bogus! */
782                                 Error(stderr,
783                                     MGET("invalid sample rate: %s\n"), s);
784                                 return (1);
785                         }
786                 }

788         }

790         *rate = irint(drate);
791         return (0);
792 }
```

unchanged_portion_omitted_