APPENDIX C: APPLICATION'S SOURCE CODE

C.1. MANAGER'S SOURCE CODE

C.1.1 "manager.h"

```
/*
 * Program name: manager.h
 * Name: Maria Jose Parajon Dominguez
 * Date: 17/08/2003
 * Description: header that has the necessary definitions for
 *              manager.c
 */

struct song {
  char songfile[Maxdat];  /*name of the file to play*/
  char songname[Maxdat];
  char artist[Maxdat];
  char album[Maxdat];
  int priority;    /*priority for playing a file*/
  int recordingtype;  /*type of formatted audio file, 0 for unknown*/
  int songnumber;  /*order number of the song in the playlist*/
  float song_length;  /*length of the song in seconds*/
  unsigned long songsize;    /*size in bytes of the song*/
  long samplerate;  /* samples per seconds */
  int audiotype;   /*type of audio file: 1 for song, 2 for audio
alerts*/
  double time_to_play;
  struct song *next_song;   /*pointer to the next song*/
  struct song *previous_song;  /*pointer to the previous song*/
} list_entry;

struct song *first;
struct song *last;
int existing_audioalert=FALSE;

void store(struct song *, struct song **, struct song **, int, struct
sockaddr_in);
void readstore(char *, int, struct sockaddr_in);
void send_message(char *, int, struct sockaddr_in);
int process_a_request(int, struct sockaddr_in);
struct song *Initialize_song_structure(struct song *);
void get_first(int, struct sockaddr_in);
void readgetnth(char *, int, struct sockaddr_in);
void saveplaylist(int, struct sockaddr_in);
void delete(char *, int, struct sockaddr_in, struct song **, struct
song **);
```

C.1.2 "manager.c"

```
/*
 * Program name: manager.c
 * Name: Maria Jose Parajon Dominguez
 * Date: 09/08/2003
 * Description: program that act as a server for pr2 client5 and
 *              player storing the audio files in the appropriate
 *              place in the playlist, deleting songs, saving the
 *              playlist in a file and sending back information about
```

```
 *               songs when requested.
 * Objective: to have a scheduler for audio running in the badge
 * Usage:./manager& from the badge
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdarg.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/time.h>

#include "playlist.h"
#include "socket.c"
#include "manager.h"

int main(void) {

 struct sockaddr_in clientaddr;
 struct sockaddr_in servaddr;

  char local_hostname[Maximum_Host_name_length];

  int listenfd;
  int exit_flag=FALSE;

  first=last=NULL;

  if (gethostname(local_hostname, Maximum_Host_name_length) == -1) {
    h_perror("could not get local host's name");
    exit(1);
  }

  listenfd=open_bind_UDP_socket(local_hostname, Audioserport);
  if ( listenfd == -1 )
    exit(1);

  do {

    exit_flag=process_a_request(listenfd, clientaddr);

    } while (!(exit_flag));

  close(listenfd);
  exit(0);

}


int process_a_request(int fd, struct sockaddr_in client){

   struct song *i;
   int z;
   int j;
```

```
   int k=FALSE;
   unsigned int clientlength;
   char     buf[Maximum_Request_Size];
   char *m;
   char r[100];
   int finish;

   memset(buf, '\0', (size_t)Maximum_Request_Size);

   clientlength=sizeof(client);

   z=recvfrom(fd, buf, sizeof(buf), 0, (struct sockaddr *)&client,
&clientlength);
   printf("%s\n", buf);
   sscanf(buf,"%d", &j);
   if(j==2){
     if(finish==TRUE){
       i=first;
       if(i==NULL){
        sprintf(r, Finish_player);
        send_message(r, fd, client);
        k=TRUE;
        return;
       }
     }
   }
   switch(j){
   case 1: readstore(buf, fd, client);
           break;
   case 2: get_first(fd, client);
           break;
   case 3: readgetnth(buf, fd, client);
           break;
   case 4: saveplaylist(fd, client);
           break;
   case 5: delete(buf,fd, client, &first, &last);
           break;
   case 6: finish=TRUE;
         break;
   default: break;
   }


   return k;
}

void send_message (char *l, int fd, struct sockaddr_in client) {

  int k;

  k=sendto(fd, l, strlen(l), 0, (struct sockaddr *)&client,
sizeof(client));

  if(k==-1){
   switch (errno){
     case EBADF:
       h_perror("An invalid descriptor was specified");
       break;
     case ENOTSOCK:
       h_perror("The argument s is not a socket");
       break;
```

```
     case EFAULT:
       h_perror("An invalid user space address was specified for a
parameter.");
       break;
     case EMSGSIZE:
       h_perror("bad message size");
       break;
       //case EAGAIN:
     case EWOULDBLOCK:
       h_perror("socket is marked non-blocking and the requested
operation would block.");
       break;
     case ENOBUFS:
       h_perror("The output queue for a network interface was full.");
       break;
     case EINTR:
       h_perror("A signal occurred.");
       break;
     case ENOMEM:
       h_perror("No memory available.");
       break;
     case EINVAL:
       h_perror("Invalid argument passed.");
       break;
     case EPIPE:
       h_perror("The local end has been shut down on a connection
oriented socket.");
       break;
     case EOPNOTSUPP:
       h_perror("not supported");
       break;
     case ENOTCONN:
       h_perror("not connected");
       break;
     case EACCES:
       h_perror("access error");
       break;
     case ENETUNREACH:
       h_perror("network unreachable");
       break;
     default:
       fprintf(stderr, "Unspecified error (h_errno=%d) in sendto\n",
errno);
     }
     printf("An error occurred while sending the datagram\n");
    exit(1);
   }
}

void readstore(char *buf, int fd, struct sockaddr_in client){

  int j=0;
  int k=0;
  struct song *i;
  char r[100];
  char move[Maxsend];
  char *p;
  char *q;


  i=(struct song *)malloc(sizeof(list_entry));
```

```
if(!i){
  sprintf(r, Out_of_memory);
  send_message(r, fd, client);
  return;
}

i=Initialize_song_structure(i);

p=strchr(buf, ' ');
q=p+1;
j=strcspn(q, " ");
while(k<j){
  i->songfile[k]=q[k];
  k++;
 }
 i->songfile[k]='\0';

 k=0;
 p=strchr(q, ':');
 q=p+1;
 p=strchr(q, ':');
 while(strcmp(q, p)!=0){
  i->songname[k]=q[0];
  k++;
  q=q+1;
 }
 k=k-7;
 i->songname[k]='\0';

 k=0;
 q=p+1;
 p=strchr(q, ':');
 while(strcmp(q,p)!=0){
  i->artist[k]=q[0];
  k++;
  q=q+1;
 }
 k=k-6;
 i->artist[k]='\0';

 k=0;
 q=p+1;
 p=strchr(q, ':');
 while(strcmp(q,p)!=0){
  i->album[k]=q[0];
  k++;
  q=q+1;
 }
 k=k-9;
 i->album[k]='\0';

 q=p+1;
 sscanf(q, "%d", &i->priority);

 p=strchr(q, ':');
 q=p+1;
 sscanf(q, "%d", &i->recordingtype);

 p=strchr(q, ':');
 q=p+1;
 sscanf(q, "%d", &i->songnumber);
```

```
   p=strchr(q, ':');
   q=p+1;
   sscanf(q, "%f", &i->song_length);

   p=strchr(q, ':');
   q=p+1;
   sscanf(q, "%u", &i->songsize);

   p=strchr(q, ':');
   q=p+1;
   sscanf(q, "%ld", &i->samplerate);

   p=strchr(q, ':');
   q=p+1;
   sscanf(q, "%d", &i->audiotype);

   sprintf(move, "cp /opt/Badge4/%s /tmp/%s", i->songfile, i->
           songfile);
   system(move);
   store(i, &first, &last, fd, client);
}


struct song *Initialize_song_structure(struct song *i) {

  strcpy(i->songfile,"file");
  strcpy(i->songname, "name");
  strcpy(i->artist, "artist");
  strcpy(i->album, "album");
  i->priority=0;
  i->recordingtype=0;
  i->songnumber=0;
  i->song_length=0;
  i->songsize=0;
  i->samplerate=0;
  i->audiotype=0;
  i->time_to_play=0;
  i->next_song=NULL;
  i->previous_song=NULL;
  return i;
}

/*function for storing the file asked to be played in the correct
order*/

void store(struct song *in, struct song **first, struct song **last,
int fd, struct sockaddr_in client){

  struct song *old, *p;
  int l;
  int k;
  time_t time_stamp=0;
  struct song *q;
  char r[100];

  if(in->audiotype==2){
    if(existing_audioalert==TRUE){
      p=*first;
      old=*last;
      while(p->audiotype!=2){
      p=p->next_song;
```

```
    }
    if(p==old){
    p->next_song=in;
    in->previous_song=p;
    in->next_song=NULL;
    in->songnumber=p->songnumber+1;
    in->time_to_play=p->time_to_play + p->song_length;
    *last=in;
      sprintf(r, File_stored);
      send_message(r, fd, client);
    return;
    }
    while(p->next_song!=NULL){
    if(p->next_song->audiotype!=2){
      in->previous_song=p;
        in->next_song=p->next_song;
        p->next_song=in;
        in->next_song->previous_song=in;
        in->songnumber=p->songnumber+1;
        in->time_to_play=p->time_to_play + p->song_length;
        q=in;
        p=in->next_song;
        while(p!=NULL){
         p->songnumber=p->songnumber+1;
         p->time_to_play=q->time_to_play + q->song_length;
         q=q->next_song;
         p=p->next_song;
        }
        sprintf(r, File_stored);
        send_message(r, fd, client);
      return;
    } else p=p->next_song;
    }
    p->next_song=in;
    in->next_song=NULL;
    in->previous_song=p;
    in->songnumber=p->songnumber+1;
    in->time_to_play=p->time_to_play + p->song_length;
    *last=in;
    sprintf(r, File_stored);
    send_message(r, fd, client);
    return;
  }
  existing_audioalert=TRUE;
}


 if (*last==NULL){          /*first song to appear */
  in->next_song=NULL;
  in->previous_song=NULL;
  in->songnumber=1;
  *last=in;
  *first=in;
  time_stamp=time(NULL);
  in->time_to_play=(double)(time_stamp);
  sprintf(r, File_stored);
  send_message(r, fd, client);
  return;
  }

 p=*first;
```

```
   old=*last;

   switch (in->priority) {
   case 1:   /*  the first in the list*/
      in->next_song=p;
      in->previous_song=NULL;
      p->previous_song=in;
      *first=in;
      time_stamp=0;
      time_stamp=time(NULL);
      in->time_to_play=(double)time_stamp;
      q=in;
      while(p!=NULL){
      p->songnumber=p->songnumber+1;
      p->time_to_play=(q->time_to_play + q->song_length);
      q=q->next_song;
      p=p->next_song;
      }
      break;
    case 2:  /* after the first one*/
      in->previous_song=p;
      if(p==old){
      in->next_song=NULL;
        *last=in;
      in->time_to_play=p->time_to_play + p->song_length;
      }
      else{
      in->next_song=p->next_song;
        in->next_song->previous_song=in;
      p=in->next_song;
      in->time_to_play=p->time_to_play + p->song_length;
      q=in;
      while(p!=NULL){
         p->songnumber=p->songnumber+1;
       p->time_to_play=q->time_to_play + q->song_length;
       q=q->next_song;
       p=p->next_song;
        }
      }
      p=*first;
      p->next_song=in;
      break;
    case 3:  /* added where specified*/
     if(in->songnumber>(old->songnumber+1)){  /* position specified is
bigger than possible */
       in->songnumber=old->songnumber+1;
      }
      k=p->songnumber;
      l=in->songnumber;
      while(k < l){
       p=p->next_song;
       k++;
      }
      if(p==NULL){    /*last in the list*/
       old->next_song=in;
        in->previous_song=old;
        in->next_song=NULL;
        in->time_to_play=old->time_to_play + old->song_length;
        *last=in;
      }
      else{
```

```
        if(p==*first){  /* first in the list*/
          in->next_song=p;
        in->previous_song=NULL;
          p->previous_song=in;
          *first=in;
        time_stamp=time(NULL);
        in->time_to_play=(double)time_stamp;
        q=in;
          while(p!=NULL){
           p->songnumber=p->songnumber+1;
         p->time_to_play=q->time_to_play + q->song_length;
         q=q->next_song;
         p=p->next_song;
         }
        }
        else {    /*included in the middle*/
          in->next_song=p;
          in->previous_song=p->previous_song;
          p->previous_song=in;
          in->previous_song->next_song=in;
        in->time_to_play= in->previous_song->time_to_play + in-
>previous_song->song_length;
        q=in;
          while(p!=NULL){
           p->songnumber=p->songnumber+1;
         p->time_to_play=q->time_to_play + q->song_length;
         q=q->next_song;
           p=p->next_song;
          }
      }
      }
      break;

    case 4:  /*last in the list*/
      in->songnumber=old->songnumber+1;
      in->time_to_play=old->time_to_play + old->song_length;
      in->next_song=NULL;
      in->previous_song=old;
      old->next_song=in;
      *last=in;
      break;
   }

   sprintf(r, File_stored);
   send_message(r, fd, client);

}

void get_first(int fd, struct sockaddr_in client) {

  struct song *i;
  char r[Maxsend];

  i=first;
  if(i==NULL){
    sprintf(r,No_files_in_playlist, 2);
    send_message(r, fd, client);
  }else {
    sprintf(r, First_file_is, i->songfile, i->recordingtype, i-
>samplerate, i->audiotype, i->song_length, i->time_to_play);
    send_message(r, fd, client);
```

```
  }

}

void readgetnth(char *buf, int fd, struct sockaddr_in client){

  char *p;
  int songnumber=0;
  struct song *i;
  char r[Maxsend];

  p=strchr(buf, ' ');
  p=p+1;
  sscanf(p, "%d", &songnumber);

  i=first;
  if(i==NULL){
   sprintf(r, No_files_in_playlist, 3);
    send_message(r, fd, client);
  }else {
   while(i!=NULL){
    if(songnumber==i->songnumber){
      sprintf(r, Nth_file_is, songnumber, i->songfile, i-
>recordingtype, i->samplerate, i->audiotype, i->song_length, i-
>time_to_play);
      send_message(r, fd, client);
      return;
    }else i=i->next_song;
   }
    sprintf(r, File_not_found, 3);
    send_message(r, fd, client);
  }
}


void saveplaylist(int fd, struct sockaddr_in client) {

  struct song *i;
  FILE *fp;
  char r[Maxsend];
  char filename[Maxsend] = "playlist";
  time_t timestamp;

  timestamp = time(NULL);
  sprintf(filename, "/tmp/playlist-%ld", (long)timestamp);

  i=first;
  if(i==NULL){
    sprintf(r, No_files_in_playlist, 4);
    send_message(r, fd, client);
    return;
  }
  fp=fopen(filename, "w");
  if(!fp){
    sprintf(r, "4.Cannot open file\n");
    send_message(r, fd, client);
    return;
  }
  while(i!=NULL){
    fprintf(fp, "%s\n%s\n%s\n%s\n%d\n%d\n%f\n%u\n%ld\n%d\n", i-
>songfile, i->songname, i->artist, i->album, i->recordingtype, i-
```

```
>songnumber, i->song_length, i->songsize, i->samplerate, i-
>audiotype);
    i=i->next_song;
  }
  sprintf(r, Playlist_saved, filename);
  send_message(r, fd, client);
  fclose(fp);

}


void delete(char *buf, int fd, struct sockaddr_in client, struct song
**first, struct song **last){

  char *p;
  char songfile[Maxdat];
  struct song *i;
  struct song *q;
  char r[Maxsend];

  p=strrchr(buf, ' ');
  p=p+1;
  sscanf(p, "%s", songfile);

  i=*first;
  if(i==NULL){
   sprintf(r, No_files_in_playlist, 5);
    send_message(r, fd, client);
    return;
  }else {
    while(i!=NULL){
      if(!strcmp(songfile,i->songfile)){
      if(i->audiotype==2){ /*if the requested song is found we check*/
        q=i;          /* it is an audio alert and if it the only one*/
        q=q->previous_song;
        if(q!=NULL){
          if(q->audiotype!=2)
            existing_audioalert=FALSE;
        }else existing_audioalert=FALSE;
      }
      if(i==*first){
        if(i==*last) *first=*last=NULL;
        else{
         q=i;
         while(i!=q){
            i->time_to_play=i->previous_song->time_to_play;
            i=i->previous_song;
         }
         i=i->next_song;
         i->previous_song=NULL;
         *first=i;
         while(i!=NULL){
           i->songnumber=i->songnumber-1;
           i=i->next_song;
         }
        }
         sprintf(r, File_deleted, songfile);
         send_message(r, fd, client);
       free(i);
         return;
      }
```

```
      if(i==*last){
        i=i->previous_song;
        *last=i;
        i->next_song=NULL;
          sprintf(r, File_deleted, songfile);
          send_message(r, fd, client);
        free(i);
          return;
      }
      q=i;
      i=*last;
      while(i!=q){
        i->time_to_play=i->previous_song->time_to_play;
        i=i->previous_song;
      }
      i->next_song->previous_song=i->previous_song;
      i->previous_song->next_song=i->next_song;
      i=i->next_song;
      while(i!=NULL){
        i->songnumber=i->songnumber-1;
        i=i->next_song;
      }
        sprintf(r, File_deleted, songfile);
        send_message(r, fd, client);
      free(i);
        return;
    }else i=i->next_song;
   }
    sprintf(r, File_not_found, 5);
    send_message(r, fd, client);
  }

}
```

C.2. USER_INTERFACE'S SOURCE CODE

C.2.1 "user_interface.h"

```
/*
 * Program name: user_interface.h
 * Name: Maria Jose Parajon Dominguez
 * Date: 17/08/2003
 * Description: program that has all the necessary definitions for
 *               user_interface.c
 */

char request[Maximum_Request_Size];
long samplerate;

int recordtype(char *);
int selectoption(void);
void enter(int, struct sockaddr_in);
void send_message(char *,int, struct sockaddr_in);
void receive_message(int, struct sockaddr_in);
void read_samplerate(FILE *, int);
void read_length(FILE *);

/* structures for chunks for wave files*/
struct RIFFChunkType {
```

```
    long RIFF;
    long NextChunkSize;
    long RIFFType;
  } RIFFChunk;

struct fmtChunkType {
    long  fmt;
    long  fmtLength;
    short WaveType;
    short Channels;
    long SampleRate;
    long BytesPerSecond;
    short BlockAlignment;
    short BitResolution;
  } fmtChunk;

/*structures for mpeg files*/

 typedef struct _ID3Info
{
    char *artist;
    char *album;
    char *title;
    char *genre;
    char *year;
    char *encoder;
    char *tracknumber;
} ID3Info;

ID3Info *read_ID3v1_tag(const char* fileName, ID3Info *info);

typedef struct id3v1_0
{
    char id[3];
    char title[30];
    char artist[30];
    char album[30];
    char year[4];
    char comment[30];
    unsigned char genre;

} id3v1_0;

void remove_trailing_spaces(char* );

void usage(void){
  printf("The range of priorities is the following:\n");
  printf("-Priority 1: audio file is played inmediately\n");
  printf("-Priority 2: audio file is the next item to be played\n");
  printf("-Priotity 3: audio file goes to the specified position in
the playlist\n");
  printf("-Priority 4: audio file goes to the end of the playlist\n");
  return;
}

#define FREE_bitrate 0
#define BAD_bitrate 0

/* bits V1,L1 V1,L2 V1,L3 V2,L1 V2,L2 & L3*/
int bitrate_table[16][5] = {
```

```
  { FREE_bitrate, FREE_bitrate, FREE_bitrate, FREE_bitrate,
FREE_bitrate}, /*0000*/
  { 32, 32, 32, 32, 8},       /* 0001 */
  { 64, 48, 40, 48, 16},      /* 0010 */
  { 96, 56, 48, 56, 24},      /* 0011 */
  { 128, 64, 56, 64, 32},     /* 0100 */
  { 160, 80, 64, 80, 40},     /* 0101 */
  { 192, 96, 80, 96, 48},     /* 0110 */
  { 224, 112, 96, 112, 56},   /* 0111 */
  { 256, 128, 112, 128, 64},  /* 1000 */
  { 288, 160, 128, 144, 80},  /* 1001 */
  { 320, 192, 160, 160, 96},  /* 1010 */
  { 352, 224, 192, 176, 112}, /* 1011 */
  { 384, 256, 224, 192, 128}, /* 1100 */
  { 416, 320, 256, 224, 144}, /* 1101 */
  { 448, 384, 320, 256, 160}, /* 1110 */
  { BAD_bitrate, BAD_bitrate, BAD_bitrate, BAD_bitrate, BAD_bitrate}
/* 1111 */
};

int samplerate_table[4][4] = {

  { 44100, 22050, 11025, -1}, /* 00 */
  { 48000, 24000, 12000, -1}, /* 01 */
  { 32000, 16000, 8000, -1},  /* 10 */
  { -1, -1, -1, -1}      /* 11 */
};


#define MPEG_version_25 0
#define MPEG_version_reserved 1
#define MPEG_version_2 2
#define MPEG_version_1 3

#define Layer_I 3
#define Layer_II 2
#define Layer_III 1
#define Layer_reserved 0

typedef struct ID3_header {
  char ident[3];
  char version[2];
  unsigned char flag;
} FORMAT_chunkP;

float song_length_ms;
float song_length_s;

unsigned long file_size;
float size_frame;
```

C.2.2 **“user_interface.c”**

```
/*
 * Program name: user_interface.c
 * Name: Maria Jose Parajón
 * Date: 9/08/2003
 * Description: Program that enables users to enter songs to be played
 *              in a playlist with certain priority, delete songs from
 *          the playlist, save the playlist in file and ask for
```

```
 *          any song store in the playlist
 * Objective: To have a program working as an interface for the user
 *            allowing this to build a playlist
 * Usage: ./user_interface from laptop
 */

#include <sys/types.h>       /* basic system data types         */
#include <sys/socket.h>      /* basic socket definitions        */
#include <netdb.h>           /* network database operations     */
#include <stdarg.h>          /* handle variable argument list   */
#include <stdlib.h>          /* standard library definitions    */
#include <netinet/in.h>      /* sockaddr_in and other Internet defns*/
#include <arpa/inet.h>       /* internet operation              */
#include <malloc.h>          /* allocate a memory block         */
#include <time.h>              /* time types  */
#include <stdio.h>             /* standard input/output routines*/
#include <unistd.h>            /* file control options    */
#include <sys/stat.h>          /* for S_xxx file mode constants */
#include <fcntl.h>             /* open(), close(), for nonblocking */
#include <string.h>            /* string operation      */
#include <errno.h>     /* assign specific positive values on error */

#include "playlist.h"
#include "socket.c"
#include "pr2.h"

int main(void) {

  int sockfd;
  struct sockaddr_in servaddr;
  struct hostent *servpointer;

 char local_hostname[Maximum_Host_name_length];

  char songfile[Maxdat];
  int songnumber=0;

  if (gethostname(local_hostname, Maximum_Host_name_length) == -1) {
    h_perror("could not get local host's name");
    exit(1);
  }

  sockfd=open_bind_UDP_socket(local_hostname, Audioprport);

  if (setup_server_sockaddr_in_structure(Audioserhost, Audioserport,
&servaddr) == -1)
    exit (1);

  for(;;){
   switch(selectoption()){
   case 1: enter(sockfd, servaddr);
           break;
   case 2: sprintf(request, Get_first_file_request);
           send_message(request, sockfd, servaddr);
           receive_message(sockfd, servaddr);
           break;
   case 3: printf("Enter the number of song requested:\n");
           scanf("%d", &songnumber);
           sprintf(request, Get_Nth_file_request, songnumber);
           send_message(request, sockfd, servaddr);
           receive_message(sockfd, servaddr);
```

```
            break;
    case 4: sprintf(request, Save_playlist_request);
            send_message(request, sockfd, servaddr);
            receive_message(sockfd, servaddr);
            break;
    case 5: printf("Enter the name of the file to delete:\n");
            scanf("%s", &songfile);
            sprintf(request, Delete_file_request, songfile);
            send_message(request, sockfd, servaddr);
            receive_message(sockfd, servaddr);
            break;
    case 6: sprintf(request, Quit_request);
            send_message(request, sockfd, servaddr);
            close(sockfd);
            exit(0);
    }
  }
}

int selectoption(void) {

  int s;

  printf("1. Store a new song in the playlist\n");
  printf("2. Get the first song in the playlist\n");
  printf("3. Get the Nth song in the playlist\n");
  printf("4. Save the playlist in a file\n");
  printf("5. Delete a song from the playlist\n");
  printf("6. Quit\n");
  do {
    printf("Enter your choice:\n");
    scanf("%d",&s);
  }while (s<0 || s>6);

    return s;
}


void enter(int sockfd, struct sockaddr_in servaddr) {

  char songfile[Maxdat];
  int priority;
  int songnumber;
  int recordingtype=0;
  int audiotype=1;
  struct stat file_status;
  unsigned long file_size;

  float song_length_ms;
  float song_length_s;
  FILE *WAVFile;
  long Before;
  long bytepers;

  FILE *MP3File;
  ID3Info *info;
  char *songname;
  char *artist;
  char *album;
  FILE *AUFile;
  for(;;){
```

```
 ent:
   songname="name";
   artist="artist";
   album="album";
   songnumber=0;
   samplerate=0;
   song_length_ms=0;
   song_length_s=0;
   file_size=0;
   bytepers=0;
   printf("Enter the file to store:\n");
   scanf("%s", songfile);
   if(!strcmp(songfile,"q")) {
     break;
   }

   printf("Enter file priority: \n");
   scanf("%d", &priority);
   switch(priority){
   case 1: songnumber=1;
           break;
   case 2: songnumber=2;
           break;
   case 3: printf("Enter the number of the file:\n");
           scanf("%d", &songnumber);
           break;
   case 4: break;
   default: usage();
           goto ent;
           break;
   }

   recordingtype=recordtype(songfile);
   if (recordingtype==0) goto ent;

    if(recordingtype==WAV_formatted_audio){

     WAVFile = fopen (songfile, "rb");
     if (!WAVFile) return;

     fseek(WAVFile, 0, SEEK_END);
     file_size = ftell(WAVFile);

     fseek(WAVFile, 0, SEEK_SET);
     fread (&RIFFChunk, sizeof (RIFFChunk), 1, WAVFile);
     if (RIFFChunk.RIFF != 0x46464952 || RIFFChunk.RIFFType !=
0x45564157) {
       fclose (WAVFile);
       return;
     }

     do {
       Before = ftell (WAVFile);
       fread (&fmtChunk, sizeof (fmtChunk), 1, WAVFile);
       fseek (WAVFile, Before + fmtChunk.fmtLength + 8, SEEK_SET);
     } while (fmtChunk.fmt != 0x20746D66);

     samplerate=fmtChunk.SampleRate;
     bytepers=fmtChunk.BytesPerSecond;
     song_length_ms=(file_size*1000)/bytepers;
     song_length_s=song_length_ms/1000;
```

```
      fclose(WAVFile);
    }

     if(recordingtype==MP3_formatted_audio){

       info = malloc(sizeof(ID3Info));
       memset(info, 0, sizeof(ID3Info));

       info=read_ID3v1_tag(songfile, info);
       songname=info->title;
       artist=info->artist;
       album=info->album;

       MP3File = fopen (songfile, "rb");
       if (!MP3File) return;

       fseek(MP3File, 0, SEEK_END);
       file_size = ftell(MP3File);

       read_samplerate(MP3File, 0);
       read_length(MP3File);
       free(info);
       fclose(MP3File);
     }

     if(recordingtype==AU_formatted_audio){

       AUFile = fopen (songfile, "rb");
       if (!AUFile) return;

       fseek(AUFile, 0, SEEK_END);
       file_size = ftell(AUFile);
       samplerate=Default_sample_rate;
       song_length_ms=(file_size*1000)/samplerate;
       song_length_s=song_length_ms/1000;
       fclose(AUFile);
     }

    sprintf(request, Store_request, songfile, songname, artist, album,
priority, recordingtype,songnumber, song_length_s, file_size,
samplerate, audiotype);

    send_message(request, sockfd, servaddr);

    receive_message(sockfd, servaddr);

  }
}

void send_message(char *buf,int fd, struct sockaddr_in servaddr){

    int k;

    k=sendto(fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
sizeof(servaddr));

    if(k==-1){
      switch (errno){
      case EBADF:
        h_perror("An invalid descriptor was specified");
        break;
```

```
     case ENOTSOCK:
       h_perror("The argument s is not a socket");
       break;
     case EFAULT:
       h_perror("An invalid user space address was specified for a
parameter.");
       break;
     case EMSGSIZE:
       h_perror("bad message size");
       break;
       //case EAGAIN:
     case EWOULDBLOCK:
       h_perror("socket is marked non-blocking and the requested
operation would block.");
       break;
     case ENOBUFS:
       h_perror("The output queue for a network interface was full.");
       break;
     case EINTR:
       h_perror("A signal occurred.");
       break;
     case ENOMEM:
       h_perror("No memory available.");
       break;
     case EINVAL:
       h_perror("Invalid argument passed.");
       break;
     case EPIPE:
       h_perror("The local end has been shut down on a connection
oriented socket.");
       break;
     case EOPNOTSUPP:
       h_perror("not supported");
       break;
     case ENOTCONN:
       h_perror("not connected");
       break;
     case EACCES:
       h_perror("access error");
       break;
     case ENETUNREACH:
       h_perror("network unreachable");
       break;
     default:
       fprintf(stderr, "Unspecified error (h_errno=%d) in sendto\n",
errno);
     }
   printf("An error ocurred  while sending the datagram\n");
    return;
   }
}

void receive_message(int fd, struct sockaddr_in servaddr){

   int servlength;
   char reci[Maximum_Response_Size];

   memset(reci,'\0', Maximum_Response_Size);

   servlength=sizeof(servaddr);
```

```
   recvfrom(fd, reci, sizeof(reci), 0, (struct sockaddr *)&servaddr,
&servlength);
   printf("%s\n", reci);

   if(!strcmp(reci,"The manager is out of memory\n")){
    close(fd);
    exit(1);
   }
}


int recordtype(char *x) {
  int j;
  int k;
  char l;
  char *p;

  k=strlen(x);
  k--;
  l=x[k];
  p=strrchr(x,'.');
  if(p==NULL) {
    j=0;
    printf("No file extension included and not stored\n");
    return j;
  }
  if(!strcmp(p,".mp3")) j=MP3_formatted_audio;
  else {
    if(!strcmp(p,".wav")) j=WAV_formatted_audio;
    else{
      if(!strcmp(p,".au")) j=AU_formatted_audio;
      else {
        j=0;
        printf("Not suitable file and has not been stored\n");
      }
    }
  }

  return j;
}

ID3Info *read_ID3v1_tag(const char* fileName, ID3Info *info)
{
    id3v1_0  id3;
    FILE  *fp;
    char   buffer[31];

    fp = fopen(fileName, "rb");
    if (fp == NULL)
        return info;

    if (fseek(fp, -128, SEEK_END))
    {
        fclose(fp);
        return info;
    }

    if (fread(&id3, 1, 128, fp) != 128)
    {
        fclose(fp);
        return info;
```

```
    }

    if(strncmp(id3.id, "TAG", 3))
    {
        fclose(fp);
        return info;
    }

    if (info == NULL)
    {
        info = malloc(sizeof(ID3Info));
        memset(info, 0, sizeof(ID3Info));
    }

    strncpy(buffer, id3.artist, 30);
    buffer[30] = 0;
    remove_trailing_spaces(buffer);
    if (strlen(buffer) && info->artist == NULL)
        info->artist = strdup(buffer);

    strncpy(buffer, id3.album, 30);
    buffer[30] = 0;
    remove_trailing_spaces(buffer);
    if (strlen(buffer) && info->album == NULL)
        info->album = strdup(buffer);

    strncpy(buffer, id3.title, 30);
    buffer[30] = 0;
    remove_trailing_spaces(buffer);
    if (strlen(buffer) && info->title == NULL)
        info->title = strdup(buffer);

    fclose(fp);
    return info;
}


void remove_trailing_spaces(char* string)
{
      char* cp = &(string[strlen(string)]);

      do
    {
          *cp = '\0';
          cp--;
      }while ((*cp == ' ') && (cp >= string));
}

void read_samplerate(FILE *MP3File, int compute_length){

  int offset;
  int bit_rate_in_KHz=0;
  int bytes=0;
  unsigned char sync;
  unsigned char sync2;
  unsigned char sync3;
  unsigned char sync4;
  int i;
  int j;
  int k;
  int id;
```

```
 int layer;
 int pad,
 float m=0;
 float length_frame=0;
 int channel;
 int number_frames=0;
 float z;

 fseek(MP3File, 0, SEEK_SET);

again:
 do{
  fread(&sync, sizeof(sync), 1, MP3File);
 }while(sync!=255);

 fread(&sync2, sizeof(sync2), 1, MP3File);

 if((sync2 & 0xE0)!= 0xE0){
   goto again;
 }

 switch(sync2 & 0x18) {
 case 0x0: id= MPEG_version_25;
         break;
 case 0x8: id= MPEG_version_reserved;
      break;
 case 0x10: id= MPEG_version_2;
      break;
 case 0x18: id= MPEG_version_1;
      break;
 }

 switch(sync2 & 0x6){
 case 0x0: layer= Layer_reserved;
           break;
 case 0x2: layer= Layer_III;
           break;
 case 0x4: layer= Layer_II;
           break;
 case 0x6: layer= Layer_I;
           break;
 }

 switch(id) {
 case MPEG_version_1:
   switch(layer) {
   case Layer_I: offset=0;
                 break;
   case Layer_II: offset=1;
                  break;
   case Layer_III: offset=2;
                   break;
   default: offset=-1;
   }
 case MPEG_version_2:
   switch(layer) {
   case Layer_I: offset=3;
                 break;
   case Layer_II: offset=4;
                  break;
   case Layer_III: offset=4;
```

```
                  break;
    default: offset=-1;
    }
  case MPEG_version_25:
    switch(layer) {
    case Layer_I: offset=3;
                  break;
    case Layer_II: offset=4;
                   break;
    case Layer_III: offset=4;
                    break;
    default: offset=-1;
    }
  }

  if(offset<0){
    fprintf(stderr, "MPEG bit rate not acceptable to this program\n");
    exit(1);
  }

  fread(&sync3, sizeof(sync3), 1, MP3File);

  switch(sync3 & 0xF0){
  case 0x0: i=0;
           break;
  case 0x10: i=1;
             break;
  case 0x20: i=2;
             break;
  case 0x30: i=3;
             break;
  case 0x40: i=4;
             break;
  case 0x50: i=5;
             break;
  case 0x60: i=6;
             break;
  case 0x70: i=7;
             break;
  case 0x80: i=8;
             break;
  case 0x90: i=9;
             break;
  case 0xA0: i=10;
             break;
  case 0xB0: i=11;
             break;
  case 0xC0: i=12;
             break;
  case 0xD0: i=13;
             break;
  case 0xE0: i=14;
             break;
  }

  bit_rate_in_KHz= bitrate_table[i][offset];

 switch(sync3 & 0xC){
    case 0x0: j=0;
              break;
    case 0x4: j=1;
```

```
            break;
    case 0x8: j=2;
            break;
    case 0xC: j=3;
            break;
    }

 switch(id){
 case 0: k=3;
        break;
 case 1: k=4;
        break;
 case 2: k=1;
        break;
 case 3: k=0;
        break;
 }

  samplerate= samplerate_table[j][k];

  if(sampling_rate==-1){
    fprintf(stderr, "MPEG sampling rate not acceptable to this
program\n");
    exit(1);
  }
  if (compute_length==0) {
    return;
  }
  else{
    switch(sync3 & 0x2){
     case 0x0: pad=0;
             break;
     case 0x2: pad=1;
             break;
    }
    switch(layer) {
     case Layer_I: m=12000*bit_rate_in_KHz;
                   bytes=(int)(((m/samplerate)+pad)*4);
                   break;
     case Layer_II: m=144000*bit_rate_in_KHz;
                  bytes=(int)((m/samplerate)+pad);
                  break;
     case Layer_III: m=144000*bit_rate_in_KHz;
                   bytes=(int)((m/samplerate)+pad);
                   break;
    }

    number_frames=(int)(size_frame/bytes);
    switch(layer) {
     case Layer_I: z=384; /* number of samples per frame */
                   break;
     case Layer_II: z=1152;
                    break;
     case Layer_III: z=1152;
                     break;
    }
    length_frame=z/samplerate;
    song_length_s=length_frame*number_frames;

    fread(&sync4, sizeof(sync4), 1, MP3File);
```

```
    switch(sync4 & 0xC0){
     case 0xC0: channel=1;
                break;
     default: channel=2;
    }
  }

}

void read_length (FILE *MP3File){

  struct ID3_header id3header;
  unsigned char extended_header[10];
  int size_id;
  int TLEN_exist=FALSE;

  unsigned char id_frame[4];
  unsigned char flag_frame[2];
  int before;
  int length_p=0;
  int length_t=0;
  int length_d=0;
  int length_tt=0;
  unsigned char each_byte;
  unsigned char length[Maxdat];
  int i;
  int j;
  int k=0;
  int l;

  fseek(MP3File, 0, SEEK_SET);

  fread(&id3header, 1, sizeof(struct ID3_header), MP3File);

  k=3;
  for(i=0; i<4; i++){
    each_byte=0;
    length_p=0;
    fread(&each_byte, 1, sizeof(each_byte), MP3File);
    length_p=(int)each_byte;
    j=7*k;
    length_p=(length_p << j);
    k=k-1;
    size_id= size_id + length_p;
  }

  if((id3header.flag & 0x40) == 0x40){
    fread(&extended_header, 1, 10, MP3File);
  }

  for(;;){
   k=0;
   memset(id_frame, '\0', sizeof(id_frame));
   length_p=0;
   length_t=0;
   before=ftell(MP3File);
   if(before > (size_id+10)) {  /* side_id + 10is the total length of
the ID3 header*/
     TLEN_exist=FALSE;
     break;
   }
```

```
   fread(&id_frame, 1, sizeof(id_frame), MP3File);
    for(i=0; i<4; i++){
     each_byte=0;
     length_p=0;
     fread(&each_byte, 1, sizeof(each_byte), MP3File);
     length_p=(int)each_byte;
     length_p=(length_p << (24-(8*k));
     k=k+1;
     length_t= length_t + length_p;
   }
   fread(&flag_frame, 1, sizeof(flag_frame), MP3File);
    if( (id_frame[0]=='T') && (id_frame[1]=='L') && (id_frame[2]=='E')
&& (id_frame[3]=='N') ) {
     TLEN_exist=TRUE;
     break;
    }
   fseek(MP3File, before+length_t+10 ,SEEK_SET);
   }
   if(TLEN_exit==TRUE) {
    k=0;
    fread(&each_byte, 1, sizeof(each_byte), MP3File); /* reading the
text encoding*/
    fread(&length, 1, (length_t-1), MP3File);
    length[length_t-1]='\0';
   }else{
    size_frame=0;
    size_frame=file_size-(128+10+size_id);
    fseek(MP3File, 0, SEEK_SET);
    read_samplerate(MP3File, 1);
   }
}
```

C.3. ALERT_GENERATOR'S SOURCE CODE

```
/*
 * Program name: alert_generator.c
 * Name: Sean Wong
 * Date: 28/5/2003
 * Description:
 *   To generate audio from a text string and send this to the audio
 *   server
 * Objective:
 *           Utilizes CMU's flite text to speech synthesis software to
 *          convert some text to a WAV file and sends the name of this
 *           file and some other information to a simple audio server.
 * Usage:
 *        programname "string"
 *        where "string" is a quoted string which is to be synthesized
 * Modified by G. Q. Maguire Jr. and Maria Jose Parajon Dominguez
 */

#include <sys/types.h>        /* basic system data types         */
#include <sys/socket.h>       /* basic socket definitions        */
#include <netdb.h>            /* network database operations     */
#include <stdarg.h>           /* handle variable argument list   */
#include <stdlib.h>          /* standard library definitions */
#include <netinet/in.h>   /* sockaddr_in and other Internet defns*/
#include <arpa/inet.h>        /* internet operation   */
#include <malloc.h>           /* allocate a memory block */
#include <time.h>            /* time types       */
```

```
#include <stdio.h>          /* standard input/output routines.  */
#include <unistd.h>            /* file control options        */
#include <sys/stat.h>       /* for S_xxx file mode constants    */
#include <fcntl.h>          /* open(), close(), for nonblocking  */
#include <string.h>         /* string operation       */
#include <errno.h>     /* assign specific positive values on error */

#include "playlist.h"
#include "socket.c"

#define Default_Sample_Rate 8000 /* sample rate to be 8000 sample per
second */
#define MAXIMUM_Text_to_Speech_Command_SIZE 1024  /* max size 1024 */
#define MAXIMUM_Text_to_Synthesize_SIZE 512         /* max size 512 */

char *program_name;

void usage()      /* generate error message "./client "text_string" */
{
  fprintf(stderr, "usage: %s \"text_string\"\n", program_name);
  exit(1);
}

int main(int argc, char **argv)
{
  int sockfd;
  struct sockaddr_in servaddr;
  struct hostent *servpointer;
  struct stat file_status;

  char local_hostname[Maximum_Host_name_length];

  long file_size;
  time_t timestamp;
  float play_time_in_ms;
  float play_time_in_s;
  int sample_rate = Default_Sample_Rate;
  char filename[Maxdat];
  char *name="name";
  char *artist="artist";
  char *album="album";
  int audiotype=2;
  char audio_request[Maximum_Request_Size];
  char text_to_speech_command[MAXIMUM_Text_to_Speech_Command_SIZE];
  char text_string[MAXIMUM_Text_to_Synthesize_SIZE];
  int k;
  int servlength;
  char reci[Maximum_Response_Size];
  char error_message[1024];

  memset(reci, '\0', Maximum_Response_Size);

  program_name = argv[0];

  if (argc == 2){
    strncpy( text_string, argv[1], MAXIMUM_Text_to_Synthesize_SIZE);
  } else {
    usage();
  }
  if (gethostname(local_hostname, Maximum_Host_name_length) == -1) {
    h_perror("could not get local host's name");
```

```
    exit(1);
  }

  sockfd=open_bind_UDP_socket(local_hostname, Audiocliport);
  if (sockfd == -1)
    exit (1);

  if (setup_server_sockaddr_in_structure(Audioserhost, Audioserport,
&servaddr) == -1)
    exit (1);

    /* Create Command */

  timestamp = time(NULL);
  sprintf(filename, "seans-audio%ld.wav", (long)timestamp);

  sprintf(text_to_speech_command, "/usr/local/flite/bin/flite -t
\"%s\" -o /opt/Badge4/%s", text_string, filename);

  system(text_to_speech_command);

  /* check the size of the file. */
  if (stat(filename, &file_status) == -1)
       {
        fprintf(stderr, "Unable to stat the file %s\n", filename);
        perror(error_message);
        exit(1);
       }
          file_size = file_status.st_size;
        play_time_in_ms = file_size/(2*8); /* Equation for play_time*/

        /* Print out the size and length of audio file */
        fprintf(stdout, "The size of the audio file: %ld bytes\n",
file_size);
        fprintf(stdout, "The length of the audio file: %f ms\n",
play_time_in_ms);

        play_time_in_s=play_time_in_ms/1000;

          /* make up the request */
        sprintf(audio_request, Store_request, filename, name, artist,
album,1, WAV_formatted_audio, 1, play_time_in_s, file_size,
Default_Sample_Rate, audiotype);

  fprintf(stdout, "audio_request = %s\n", audio_request);


   k=sendto(sockfd, audio_request, strlen(audio_request), 0, (struct
sockaddr *)&servaddr, sizeof(servaddr));

   if(k==-1){
    switch (errno){
     case EBADF:
       h_perror("An invalid descriptor was specified");
       break;
     case ENOTSOCK:
       h_perror("The argument s is not a socket");
       break;
     case EFAULT:
       h_perror("An invalid user space address was specified for a
parameter.");
```

```
       break;
     case EMSGSIZE:
       h_perror("bad message size");
       break;
       //case EAGAIN:
     case EWOULDBLOCK:
       h_perror("socket is marked non-blocking and the requested
operation would block.");
       break;
     case ENOBUFS:
       h_perror("The output queue for a network interface was full.");
       break;
     case EINTR:
       h_perror("A signal occurred.");
       break;
     case ENOMEM:
       h_perror("No memory available.");
       break;
     case EINVAL:
       h_perror("Invalid argument passed.");
       break;
     case EPIPE:
       h_perror("The local end has been shut down on a connection
oriented socket.");
       break;
     case EOPNOTSUPP:
       h_perror("not supported");
       break;
     case ENOTCONN:
       h_perror("not connected");
       break;
     case EACCES:
       h_perror("access error");
       break;
     case ENETUNREACH:
       h_perror("network unreachable");
       break;
     default:
       fprintf(stderr, "Unspecified error (h_errno=%d) in sendto\n",
errno);
     }
    printf("An error occurred while sending the datagram\n");
    return;
   }

   servlength=sizeof(servaddr);

   recvfrom(sockfd, reci, sizeof(reci), 0, (struct sockaddr*)
            &servaddr, &servlength);
   printf("%s\n", reci);

  close(sockfd);
  exit(0);
}
```

C.4. PLAYER'S SOURCE CODE

C.4.1 “player.h”

```
/*
 * Program name: player.h
 *
 * Name: Maria Jose Parajon Dominguez
 *
 * Date: 17/08/2003
 *
 * Description: programs that has the necessary definitions for
 *  player.c
 *
 */

#define AUDIO_DEVICE "/dev/sound/dsp"
#define MAX_number_of_bytes_per_read 8192

/*
 * WAV format according to:
 *
 * http://www.technology.niagarac.on.ca/courses/comp630/WavFileFormat.
 * html
 *
 * RIFF Chunk (12 bytes in length total)
 * Byte Number
 * 0 - 3    "RIFF" (ASCII Characters)
 * 4 - 7    Total Length Of Package To Follow (Binary, little endian)
 * 8 - 11   "WAVE" (ASCII Characters)
 */

typedef struct RIFF_chunk {
  char riff[4];
  int  total_length;         /* this is in little endian order */
  char wave[4];
} RIFF_chunkP;

/*
 * FORMAT Chunk (24 bytes in length total)
 * Byte Number
 * 0 - 3    "fmt " (ASCII Characters) Note this is 'f' 'm' 't' 'space'
 * 4 - 7    Length Of FORMAT Chunk (Binary, always 0x10)
 * 8 - 9    Always 0x01
 * 10 - 11  Channel Numbers (Always 0x01=Mono, 0x02=Stereo)
 * 12 - 15      Sample Rate (Binary, in Hz)
 * 16 - 19  Bytes Per Second
 * 20 - 21  Bytes Per Sample: 1=8 bit Mono, 2=8 bit Stereo or 16 bit
 * Mono, 4=16 bit Stereo
 * 22 - 23  Bits Per Sample
 */

typedef struct FORMAT_chunk {
  char fmt[4];
  int length_of_FORMAT_chunk; /* always 0x10 */
  short int marker;           /* always 0x01 */
  short int channel_numbers;  /* (Always 0x01=Mono, 0x02=Stereo) */
  int sample_rate;            /* (Binary, in Hz) */
  int bytes_per_second;
  short int bytes_per_sample; /* 1=8 bit Mono, 2=8 bit Stereo or 16
bit Mono, 4=16 bit Stereo */
  short int bits_per_sample;
```

```
} FORMAT_chunkP;

/*
 * DATA Chunk
 * Byte Number
 * 0 - 3    "data" (ASCII Characters)
 * 4 - 7    Length Of Data To Follow
 * 8 - end  Data (Samples)
 */

typedef struct DATA_Chunk_header {
  char marker[4];
  int length;
} DATA_Chunk_headerP;

char request[Maximum_Request_Size];

int recordingtype=0;
char songfile[Maxdat];
long samplerate=0;
int audiotype=0;
float songlength=0;
float timetoplay=0;

int exit_flag=FALSE;

void send_message(char *,int, struct sockaddr_in);
int receive_message(int, struct sockaddr_in);
int player_sleep(int);
void play_song(void);
int get_delete_1th_song(int, struct sockaddr_in server);
void sayfile(char *, int);
void playwav(char *, int);

int NEED_TO_DO_STEREO_CONVERSION = FALSE;


/* from vovida's g711.c source file */
/*
 * g711.c
 *
 * u-law, A-law and linear PCM conversions.
 */
#define     SIGN_BIT    (0x80)           /* Sign bit for a A-law byte.
*/
#define     QUANT_MASK  (0xf)       /* Quantization field mask. */
#define     NSEGS       (8)         /* Number of A-law segments. */
#define     SEG_SHIFT   (4)      /* Left shift for segment number. */
#define     SEG_MASK    (0x70)           /* Segment field mask. */
#define     BIAS        (0x84)           /* Bias for linear code. */

#if 1
/*
** This routine converts from ulaw to 16 bit linear.
**
** Craig Reese: IDA/Supercomputing Research Center
** 29 September 1989
**
** References:
** 1) CCITT Recommendation G.711  (very difficult to follow)
** 2) MIL-STD-188-113,"Interoperability and Performance Standards
```

```
**     for Analog-to_Digital Conversion Techniques,"
**     17 February 1987
**
** Input: 8 bit ulaw sample
** Output: signed 16 bit linear sample
*/

int ulaw2linear (unsigned char ulawbyte){

  static const int exp_lut[8] = { 0, 132, 396, 924, 1980, 4092, 8316,
16764 };
  int sign, exponent, mantissa, sample;

  ulawbyte = ~ ulawbyte;
  sign = ( ulawbyte & 0x80 );
  exponent = ( ulawbyte >> 4 ) & 0x07;
  mantissa = ulawbyte & 0x0F;
  sample = exp_lut[exponent] + ( mantissa << ( exponent + 3 ) );
  if ( sign != 0 ) sample = -sample;

  return sample;
}

#else

/*
 * ulaw2linear() - Convert a u-law value to 16-bit linear PCM
 *
 * First, a biased linear code is derived from the code word. An
unbiased
 * output can then be obtained by subtracting 33 from the biased code.
 *
 * Note that this function expects to be passed the complement of the
 * original code word. This is in keeping with ISDN conventions.
 */
int
ulaw2linear(
    unsigned char u_val)
{
    int     t;

    /* Complement to obtain normal u-law value. */
    u_val = ~u_val;

    /*
     * Extract and bias the quantization bits. Then
     * shift up by the segment number and subtract out the bias.
     */
    t = ((u_val & QUANT_MASK) << 3) + BIAS;
    t <<= ((unsigned)u_val & SEG_MASK) >> SEG_SHIFT;

    return ((u_val & SIGN_BIT) ? (BIAS - t) : (t - BIAS));
}
#endif

int set_audio_sample_rate(int audiofd, int sample_rate)
{
  int ioctlParam;
  int ioctlstatus;

  ioctlParam = sample_rate;
```

```
  if((ioctlstatus=ioctl(audiofd, SNDCTL_DSP_SPEED, &ioctlParam))==-1 )
{
    perror("SNDCTL_DSP_SPEED");
    close(audiofd);
    return(-1);
  }

  ioctlParam = 0;
  if((ioctlstatus=ioctl( audiofd, SNDCTL_DSP_STEREO, &ioctlParam)) ==
-1 ) {
    perror("SNDCTL_DSP_STEREO");
    close(audiofd);
    return(-1);
  }

  if (ioctlParam < 0)
   NEED_TO_DO_STEREO_CONVERSION=TRUE;
   ioctlParam = sample_rate;
  if((ioctlstatus=ioctl( audiofd, SOUND_PCM_WRITE_RATE, &ioctlParam))
== -1 ) {
    perror("SOUND_PCM_WRITE_RATE");
    close(audiofd);
    return(-1);
  }
}

int open_audio(char *device, int sample_rate)
{
    int audiofd;

   audiofd = open( device, O_WRONLY | O_NDELAY );
   if ( audiofd < 0 ) {
      perror( "opening audio device" );
       return(-1);
    }

   set_audio_sample_rate(audiofd, sample_rate);

   return (audiofd);
}
```

C.4.2 “player.c”

```
/*
 * Program name: player.c
 * Name: Maria Jose Parajon Dominguez
 * Date: 09/08/2003
 * Description: programs that plays the songs in the playlist
 *              everytime there is one stored
 * Objective: to test the functioning of the audio in the badge
 * Usage: ./player& from the badge
 */

#include <sys/types.h>       /* basic system data types         */
#include <sys/socket.h>      /* basic socket definitions        */
#include <netdb.h>           /* network database operations     */
#include <stdarg.h>          /* handle variable argument list   */
#include <stdlib.h>        /* standard library definitions      */
#include <netinet/in.h>   /* sockaddr_in and other Internet defns*/
#include <arpa/inet.h>       /* internet operation              */
```

```
#include <malloc.h>          /* allocate a memory block         */
#include <time.h>           /* time types               */
#include <stdio.h>         /* standard input/output routines. */
#include <unistd.h>            /* file control options    */
#include <sys/stat.h>       /* for S_xxx file mode constants   */
#include <fcntl.h>         /* open(), close(), for nonblocking     */
#include <string.h>            /* string operation         */
#include <errno.h>     /* assign specific positive values on error */
#include <sys/soundcard.h>

#include "playlist.h"
#include "socket.c"
#include "player.h"

int main(void) {

  int sockfd;
  struct sockaddr_in servaddr;
  struct hostent *servpointer;

  int count=0;

  char local_hostname[Maximum_Host_name_length];

  int flag=0;
  int k;

  if (gethostname(local_hostname, Maximum_Host_name_length) == -1) {
    h_perror("could not get local host's name");
    exit(1);
  }

  sockfd=open_bind_UDP_socket(local_hostname, Audioplayport);
  if (sockfd == -1)
    exit (1);

  if (setup_server_sockaddr_in_structure(Audioserhost, Audioserport,
&servaddr) == -1)
    exit (1);

  do{
   flag=get_delete_1th_song(sockfd, servaddr);
   if(flag==1){
     play_song();
   }else {
     count=player_sleep(count);
   }
  } while(!exit_flag);
  close(sockfd);
}


void send_message(char *buf, int fd, struct sockaddr_in servaddr){

   int k;

   k=sendto(fd, buf, strlen(buf), 0, (struct sockaddr *)&servaddr,
sizeof(servaddr));

   if(k==-1){
    switch (errno){
```

```
     case EBADF:
       h_perror("An invalid descriptor was specified");
       break;
     case ENOTSOCK:
       h_perror("The argument s is not a socket");
       break;
     case EFAULT:
       h_perror("An invalid user space address was specified for a
parameter.");
       break;
     case EMSGSIZE:
       h_perror("bad message size");
       break;
       //case EAGAIN:
     case EWOULDBLOCK:
       h_perror("socket is marked non-blocking and the requested
operation would block.");
       break;
     case ENOBUFS:
       h_perror("The output queue for a network interface was full.");
       break;
     case EINTR:
       h_perror("A signal occurred.");
       break;
     case ENOMEM:
       h_perror("No memory available.");
       break;
     case EINVAL:
       h_perror("Invalid argument passed.");
       break;
     case EPIPE:
       h_perror("The local end has been shut down on a connection
oriented socket.");
       break;
     case EOPNOTSUPP:
       h_perror("not supported");
       break;
     case ENOTCONN:
       h_perror("not connected");
       break;
     case EACCES:
       h_perror("access error");
       break;
     case ENETUNREACH:
       h_perror("network unreachable");
       break;
     default:
       fprintf(stderr, "Unspecified error (h_errno=%d) in sendto\n",
errno);
     }
     printf("An error occurred while sending the datagram\n");
    exit(1);
   }
}

int receive_message(int fd, struct sockaddr_in servaddr){

   unsigned int servlength;
   char dat[Maxdat];
   char reci[Maximum_Response_Size];
   char *p;
```

```
   char *q;
   int i=0;
   int j=0;
   int k=0;
   char move[Maxdat];
   float timestamp=0;

   memset(reci, '\0', Maximum_Response_Size);
   recordingtype=0;
   memset(songfile, '\0', Maxdat);
   samplerate=0;
   audiotype=0;
   songlength=0;
   timetoplay=0;
   memset(dat, '\0', Maxdat);

   servlength=sizeof(servaddr);

   recvfrom(fd, reci, sizeof(reci), 0, (struct sockaddr *)&servaddr,
&servlength);
  printf("%s\n", reci);

  if(!strcmp(reci, Finish_player)){
    exit_flag=TRUE;
    return i;
  }
  sprintf(dat, No_files_in_playlist, 2);
  if (!strcmp(reci, dat)){
   return i;
  }else {

  p=strchr(reci, ':');
  q=p+1;
  j=strcspn(q, " ");
  while(k<j){
    songfile[k]=q[k];
    k++;
  }
  songfile[k]='\0';

  p=strchr(q, ':');
  q=p+1;
  sscanf(q, "%d", &recordingtype);

  p=strchr(q, ':');
  q=p+1;
  sscanf(q, "%ld", &samplerate);

  p=strchr(q, ':');
  q=p+1;
  sscanf(q, "%d", &audiotype);

  p=strchr(q, ':');
  q=p+1;
  sscanf(q, "%f", &songlength);

  p=strchr(q, ':');
  q=p+1;
  sscanf(q, "%f", &timetoplay);

  timestamp =(float) time(NULL);
```

```
  if(timetoplay>timestamp){
    return i;
  }else {
   i=1;
   return i;
   }
 }
}

int player_sleep(int count){

  count++;
  sleep(2);
  return count;

}

void play_song(void){

  char plaympeg[Maxdat];
  char remove[Maxdat];
  int audiofd;

  switch(recordingtype) {
  case MP3_formatted_audio:
    sprintf(plaympeg, "mpg123 /tmp/%s", songfile);
    system(plaympeg);
    break;
  case WAV_formatted_audio:
    audiofd=open_audio(AUDIO_DEVICE, samplerate);
    if (audiofd) {
      playwav(songfile, audiofd);
      close(audiofd);
    }
    if(audiotype==2){
      sprintf(remove, "rm /tmp/%s", songfile);
      system(remove);
    }
    break;
  case AU_formatted_audio:
    audiofd=open_audio(AUDIO_DEVICE, samplerate);
    if (audiofd) {
      sayfile(songfile, audiofd);
      close(audiofd);
    }
    break;
  default:
    printf("unknown file type\n");
  }
}

int get_delete_1th_song(int fd, struct sockaddr_in server){

  int j=0;
  unsigned int servlength;
  char reci[Maximum_Response_Size];

  memset(reci, '\0', Maximum_Response_Size);

  sprintf(request, Get_first_file_request);
```

```
  send_message(request, fd, server);
  j=receive_message(fd, server);
  if(j==1){
   sprintf(request, Delete_file_request, songfile);
   send_message(request, fd, server);

   servlength=sizeof(server);

   recvfrom(fd, reci, sizeof(reci), 0, (struct sockaddr *)&server,
&servlength);
   printf("%s\n", reci);
  }

  return j;
}

void sayfile(char *filename, int audiofd) {

    int filefd;
    int r, w;
    unsigned char buf[1024];
    unsigned short buf2[1024];
    unsigned short buf4[2048];     /* for use when the device does not
do mono-to-stereo conversion */
    char pathname[200];
    int i;
    unsigned short audio_sample;
    int write_trial;

    (void) sprintf( pathname, "/tmp/%s", filename );
    filefd = open( pathname, O_RDONLY );
    if ( filefd < 0 )
      {
      perror( "opening audio file" );
      exit( 1 );
      }

    for ( ; ; ) {
      r = read( filefd, buf, sizeof(buf) );
      if ( r < 0 )
          {
          perror( "reading from audio file" );
          exit( 1 );
          }
      if ( r == 0 )
          break;

      for (i=0; i < r; i++) {
        audio_sample=ulaw2linear(buf[i]);

        if (NEED_TO_DO_STEREO_CONVERSION) {
          buf4[2*i]=audio_sample;
          buf4[2*i+1]=audio_sample;
        }
        else buf2[i]=audio_sample;

      }

      for (write_trial=0; write_trial < 8; write_trial++ ) {
        if (NEED_TO_DO_STEREO_CONVERSION)
          w = write( audiofd, buf4, 4*r );
```

```
        else w = write( audiofd, buf2, 2*r );

        if ( w > 0 ) break;
        usleep( 100000 );
      }
      if ( w < 0 ) {
        perror( "writing to audio device" );
        exit( 1 );
      }
    }

    close( filefd );
}

void playwav(char *filename, int audiofd)
{
    int filefd;
    int r, w;

              /* define the input buffer so that it can be accessed as
either bytes or shorts.*/
    union u_tag {
      unsigned char buf[MAX_number_of_bytes_per_read];
      unsigned short sbuf[MAX_number_of_bytes_per_read/2];
    } input_buffer;

    union u_tag2 {
      unsigned char buf[4*MAX_number_of_bytes_per_read];
      unsigned short sbuf[2*MAX_number_of_bytes_per_read];
    } output_buffer;

    unsigned short buf2[2048];
    unsigned short *buf_as_short;

    char pathname[200];
    int i, j;
    int write_trial;

    struct RIFF_chunk RIFF_header;
    struct FORMAT_chunk FORMAT_header;
    struct DATA_Chunk_header DATA_header;

     int total_length_RIFF;

     int number_of_channels;    /* 1=Mono, 2=Stereo */
     int sample_rate;           /* in Hz */
     int bytes_per_sample;      /* 1=8 bit Mono, 2=8 bit Stereo or 16
bit Mono, 4=16 bit Stereo */
     int bytes_per_second;

     int data_chunk_length;
     int bytes_left_in_chunk;  /* remaining bytes to process in a data
chunk */

     unsigned short int testing;

    (void) sprintf( pathname, "/tmp/%s", filename );
    filefd = open( pathname, O_RDONLY );
    if ( filefd < 0 )
        {
        perror( "opening audio file" );
```

```
        exit( 1 );
        }

    r = read( filefd, &RIFF_header, sizeof(struct RIFF_chunk) );
    if ( r < sizeof(struct RIFF_chunk) ) {
      perror( "reading WAV audio file header" );
      exit( 1 );
    }

                        /* check the RIFF header */
    if ( !( (RIFF_header.riff[0]=='R') && (RIFF_header.riff[1]=='I')
&&
          (RIFF_header.riff[2]=='F') && (RIFF_header.riff[3]=='F') &&
          (RIFF_header.wave[0]=='W') && (RIFF_header.wave[1]=='A') &&
          (RIFF_header.wave[2]=='V') && (RIFF_header.wave[3]=='E')
          ) ) {
       perror( "this is not a RIFF header" );
    }
total_length_RIFF =  RIFF_header.total_length;

    r = read( filefd, &FORMAT_header, sizeof(struct FORMAT_chunk) );
    if ( r < sizeof(struct FORMAT_chunk)) {
      perror( "reading WAV audio file FORMAT information" );
      exit( 1 );
    }
                                /* check the RIFF header */
    if ( !( (FORMAT_header.fmt[0]=='f') && (FORMAT_header.fmt[1]=='m')
&&
          (FORMAT_header.fmt[2]=='t') && (FORMAT_header.fmt[3]==0x20)
&&
          (FORMAT_header.length_of_FORMAT_chunk == 0x10) &&
          (FORMAT_header.marker == 0x01)
          ) ) {
       perror( "this is not a RIFF FORMAT header" );
    }

    number_of_channels = FORMAT_header.channel_numbers;
    sample_rate = FORMAT_header.sample_rate;
    bytes_per_sample = FORMAT_header.bytes_per_sample;
    bytes_per_second =  FORMAT_header.bytes_per_second;

set_audio_sample_rate(audiofd, sample_rate);

 for(;;) {
    r = read( filefd, &DATA_header, sizeof(struct DATA_Chunk_header)
);
    if ( r < 0 ) {
      perror( "reading from data chunk header" );
      exit( 1 );
    }
    if ( r == 0 )
      break;

                                /* check data chunk header */
    if ( !( (DATA_header.marker[0] == 'd') &&
            (DATA_header.marker[1] == 'a') &&
            (DATA_header.marker[2] == 't') &&
            (DATA_header.marker[3] == 'a') ) ) {
      perror( "this is not a RIFF DATA header" );
    }
```

```
data_chunk_length = DATA_header.length;

bytes_left_in_chunk = data_chunk_length;

while (bytes_left_in_chunk) {

  r = read( filefd, input_buffer.buf, sizeof(input_buffer.buf) );
  if ( r < 0 ) {
    perror( "reading from audio file" );
    exit( 1 );
  }
  if ( r == 0 )
    return;
                  /*  update amount left to read in this chunk */
  bytes_left_in_chunk = bytes_left_in_chunk - r;

  switch (bytes_per_sample) {
  case 4:                   /* two 16 bit stereo samples */
    do  {
      w = write( audiofd, input_buffer.buf, r );
    } while ( w < 0 );

    break;

  case 2: /* two 8 bit stereo samples or one 16 bit mono sample */
    if (number_of_channels == 1) {
                            /* a single 16 bit mono sample */
      if (NEED_TO_DO_STEREO_CONVERSION) {
        for (i=0; i < (r/2); i=i+1) {
          testing = input_buffer.sbuf[i];
          output_buffer.sbuf[2*i] = testing;
          output_buffer.sbuf[(2*i)+1] = testing;
        }

        do  {
          w = write( audiofd, output_buffer.sbuf, 2*r);

        } while ( (w < 0) && (errno == EAGAIN));
        usleep(100000);

      } else {
        do  {
          w = write( audiofd, input_buffer.buf, r );
        } while ( w < 0 );
      }
    } else {          /* two 8 bit stereo samples */
      for (i=0; i < r; i++) {
        buf2[i]=input_buffer.buf[i];
      }
      do  {
        w = write( audiofd, buf2, 2*r );
      } while ( w < 0 );
    }
    break;

  case 1:                   /* a single 8 bit mono sample */
    for (i=0; i < r; i++) {
      buf2[2*i+1]=buf2[2*i]=input_buffer.buf[i];
    }
    do  {
      w = write( audiofd, buf2, 4*r );
```

```
        } while ( w < 0 );
        break;

      default:
        perror( "error in number of channels" );
        exit(1);
      }

    }
  }

  close(filefd);
}
```

C.5. SOURCE CODE OF “socket.c”

```
/*
 *
 * Program name: socket.c
 *
 * Name: Maria Jose Parajon Dominguez
 * Date: 2003.08.12
 *
 * Description: useful socket and other utility function(s)
 *
 * Objective:   simplify the playlist manager's clients
 *
 * Modified 2003.08.12 by G. Q. Maguire Jr.
 *          renamed function and added another parameter
 *          added additional error handling
 *
 */
#include <sys/types.h>        /* basic system data types         */
#include <sys/socket.h>       /* basic socket definitions        */
#include <netdb.h>            /* network database operations     */
#include <stdarg.h>           /* handle variable argument list   */
#include <stdlib.h>         /* standard library definitions      */
#include <netinet/in.h>    /* sockaddr_in and other Internet defns */
#include <arpa/inet.h>       /* internet operation               */
#include <stdio.h>            /* standard input/output routines.  */
#include <unistd.h>           /* file control options */
#include <string.h>           /* string operation  */
#include <errno.h>      /* assign specific positive values on error */

void h_perror(const char *s)
{
  fprintf(stderr, "%s\n", s);
}

/* function for opening and binding an UDP socket */
int open_bind_UDP_socket(char *hostname, unsigned short port_number){

 int sockfd;
 struct sockaddr_in cliaddr;
 struct hostent *clipointer;

 sockfd = socket(AF_INET, SOCK_DGRAM, 0);

 bzero(&cliaddr, sizeof(cliaddr));
 cliaddr.sin_family = AF_INET;
```

```
 cliaddr.sin_port = htons(port_number);

 if ( (clipointer = gethostbyname(hostname)) ) {
  memcpy(&cliaddr.sin_addr.s_addr, *((struct in_addr **)clipointer->
         h_addr_list),sizeof(struct in_addr));
 } else {
    //  Process the error if gethostbyname() is not successful
    switch (errno){
    case NETDB_INTERNAL:
      h_perror("Internal error gethostbyname");
      break;
    case HOST_NOT_FOUND:
      h_perror("Not able to find host");
      break;
    case TRY_AGAIN:
      h_perror("Please try again");
      break;
    case NO_RECOVERY:
      h_perror("No recovery");
      break;
    case NO_DATA:
      h_perror("No data");
      break;
    default:
      h_perror("Unspecified error in gethostbyname()");
    }
    exit(1);
 }

#ifdef DEBUG
  fprintf(stdout, "%s\n", inet_ntoa(cliaddr.sin_addr));
#endif

 if ( bind(sockfd, (struct sockaddr *)&cliaddr, sizeof(cliaddr)) == 1)
 {
     //  Process the error if bind() is not successful
    switch (errno){
    case EBADF:
      h_perror("socket is not a valid descriptor");
      break;
    case EINVAL:
      h_perror("invalid argument");
      break;
    case EACCES:
      h_perror("address is protected/permission error");
      break;
    case ENOTSOCK:
      h_perror("argument is not a socket descriptor");
      break;
    case EROFS:
      h_perror("read only FS error");
      break;
    case EFAULT:
      h_perror("address structure is outside user's address space");
      break;
    case ENAMETOOLONG:
      h_perror("name is too long");
      break;
    case ENOENT:
      h_perror("file does not exist");
      break;
```

```
    case ENOMEM:
      h_perror("insufficient kernel memory");
      break;
    case ENOTDIR:
      h_perror("component of the path is not a directory");
      break;
    case ELOOP:
      h_perror("too many symbolic links");
      break;
    case EADDRINUSE:
      h_perror("Address already in use");
      break;

    default:
      fprintf(stderr, "Unspecified error (errno=%d) in bind()",
errno);
    }
    return (-1);
  }

 return sockfd;
}


int setup_server_sockaddr_in_structure(char *server_name, unsigned
short port_number, struct sockaddr_in *servaddr)
{
  struct hostent *servpointer;

  bzero(servaddr, sizeof(struct sockaddr_in));
  servaddr->sin_family = AF_INET;
  servaddr->sin_port = htons(port_number);

  if ( (servpointer = gethostbyname(server_name)) ) {
    memcpy(&(servaddr->sin_addr), *((struct in_addr **)servpointer->
            h_addr_list), sizeof(struct in_addr));

  } else {
    //  Process the error if gethostbyname is not successful
    switch (h_errno){
    case NETDB_INTERNAL:
      h_perror("Internal error gethostbyname");
      break;
    case HOST_NOT_FOUND:
      h_perror("Not able to find host");
      break;
    case TRY_AGAIN:
      h_perror("Please try again");
      break;
    case NO_RECOVERY:
      h_perror("No recovery");
      break;
    case NO_DATA:
      h_perror("No data");
      break;
    default:
      h_perror("Unspecified error in gethostbyname");
    }
    return(-1);
  }
```

```
  return(1);   /* successfully filled in the sockadd_in structure */
}
```

C.6. SOURCE CODE OF “playlist.h”

```
#define FALSE 0
#define TRUE 1
                            /* the port server will be connecting to */
#define Audioserport  50555
                    /* the port user_interface will be connecting to */
#define Audioprport   50556
                            /* the port player will be connecting to */
#define Audioplayport 50557
                   /* the port alert_generator will be connecting to */
#define Audiocliport  50558

#define Audioserhost "badge41"

#define Maximum_Host_name_length 256

                                /* LISTENQ as 10 */
#define LISTENQ 10
                            /* maximum line will be 1024  */
#define MAXLINE 1024
#define Maximum_Request_Size 5000
#define Maximum_Response_Size 5000
#define Maxdat 100
#define Maxsend 1000

#define Get_first_file_request "2.Get first file\n"
#define Get_Nth_file_request "3.Get %dth file\n"
#define Save_playlist_request "4.Save playlist in file\n"
#define Delete_file_request "5.Delete the file %s\n"
#define Quit_request "6.Quit\n"
#define Store_request "1.Store %s songname:%s artist:%s album:%s
priority:%d recordingtype:%d songnumber:%d songlength:%f songsize:%ld
samplerate:%ld audiotype:%d;\n"

#define Out_of_memory "1.The manager is out of memory\n"
#define File_stored "1.The file was stored\n"
#define No_files_in_playlist "%d.There are no files in the playlist
yet\n"
#define File_not_found "%d.The requested file was not found"
#define Finish_player "6.Finish the player\n"

#define First_file_is "2.The first file to play is:%s with
recordingtype:%d samplerate:%ld audiotype:%d songlength:%f
timetoplay:%f\n"
#define Nth_file_is "3.The %dth file to play is:%s with
recordingtype:%d samplerate:%ld audiotype:%d songlength:%f
timetoplay:%f\n"
#define Playlist_saved "4.Playlist saved in file %s\n"
#define File_deleted "5.The %s file has been deleted\n"

#define MP3_formatted_audio 1
#define WAV_formatted_audio 2
#define AU_formatted_audio 3

#define Default_sample_rate 8000
```