

```

/* interplay.c -- execute me to compile me
lc -bl -cfist -v -j73 interplay.c
blink FROM lc:/lib/c.o + interplay.o TO interplay LIBRARY
lc:/lib/lc.lib+lib:amiga.lib
quit
*/

/* Interplay.c - Run from Shell (CLI) only. Given two file names of IFF
** 8SVX 8-bit sampled audio data, plays the data from both files using just
** one channel. This demonstrates how virtual audio channels can be
** implemented.
**
** The program supports two different methods for virtual voices. Method 1
** (the default method) interleaves bytes from each file so that the data words
** fed into the Amiga's audio hardware contain one byte each from the given
** files. The samples are then played back at twice their normal speed. Since
** each sample only gets half of the playback bandwidth, the speed sounds
** correct. To the listener, it sounds as if both samples are playing
** simultaneously even though only one channel is used.
**
** Normally the maximum playback rate with the Amiga's audio hardware is about
** 28K bytes/sec. Since interleaving requires doubling the nominal sampling
** rate, it will only work with audio data created at a sampling rate of 14K
** bytes/sec or less.
**
** Method 2, takes one byte from each file, sums them and divides by two.
** The resulting byte value is sent to the Amiga's audio hardware. No speed
** increase is required for this technique, however some noise is introduced
** by the averaging of the byte values. To use method 2, include the SUM
** keyword as the last argument typed on the command line. Examples:
**
**     interplay talk.8svx music.8svx SUM (Uses method 2, averaging)
**     interplay talk.8svx music.8svx      (Uses method 1, interleaving)
**     interplay talk.8svx                  (Normal single file 8SVX playback)
**
** For an example of conventional IFF 8SVX audio see the "Amiga ROM Kernel
** Reference Manual: Devices", 3rd edition (ISBN 0-201-56775-X), page 28 and
** page 515.
*/

#include <exec/types.h>
#include <exec/devices.h>
#include <exec/memory.h>
#include <devices/audio.h>
#include <dos/dos.h>

#include <iff/iff.h>
#include <iff/8svx.h>

#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/alib_protos.h>

#include <stdio.h>
#include <string.h>
#include <dos.h>      /* This is the dos.h file from SAS/C not Commodore */

#ifdef SAS_C
int CXBRK(VOID) { return(0); };
int chkabort(VOID) { return(0); };
#endif

```

```

#define BUF_SIZE 1024

/* Prototypes for functions defined in this program */
struct IOAudio *SieveChannel( VOID );
VOID ReleaseChannel( struct IOAudio * );
char *Parse8svx(char *, struct InterPlay * );
VOID EndParse( struct InterPlay * );
VOID FillAudio(struct InterPlay *, struct IOAudio * );

struct InterPlay
{
    ULONG      sample_done; /* This is the main structure used for */
    UBYTE      *sample_byte; /* storage of playback state info. */
    UBYTE      *sample_loc; /* 0=Keep playing, 1=all done playing. */
    ULONG      sample_size; /* Pointer for going through the data. */
    struct InterPlay *next_iplay; /* Start of 8SVX BODY data in memory. */
                                /* and total size of file for freeing. */
    LONG        offsetBody; /* Link to second data set. NULL means */
    UWORD       sample_speed; /* no second file name was given. */
    BOOL        USE_SUMMING; /* Offset into the file of BODY Chunk. */
                                /* Value for audio period register. */
                                /* TRUE means use averaging, */
                                /* FALSE means use interleaving. */
};

/* Version string for AmigaDOS VERSION command. */
UBYTE versiontag[] = "$VER: Interplay 1.0 (2.2.93)";

/*-----
**
**     main()
**
**-----
*/

VOID main(int argc, char **argv)
{
    struct InterPlay mainplay, otherplay; /* Two instances of the InterPlay */
                                          /* structure, one for each file. */
    struct IOAudio *pIOA_1=NULL,
                  *pIOA_2=NULL,          /* Two IOAudio pointers, plus one */
                  *pIOA   =NULL;         /* for switching back and forth */
    struct MsgPort *mport1=NULL,        /* during double-buffering. */
                  *mport2=NULL,         /* Two MsgPort pointers, plus one */
                  *mport   =NULL;       /* for switching back and forth. */

    struct Message *msg;                 /* For the GetMsg() call. */

    LONG aswitch = 0L;                   /* Double-buffering logical switch. */

    static BYTE chip playbuffer1[BUF_SIZE]; /* Two buffers, one for each IOAudio */
    static BYTE chip playbuffer2[BUF_SIZE]; /* request. Play out of one while */
                                          /* the other is being set up. */

    char *errormsg; /* For error returns */
    ULONG wakemask=0L; /* For Wait() call */

    /* Give an AmigaDOS style help message */
    if( (argc == 2) && !strcmp(argv[1], "?\0") )
        printf("8SVX-FILES/M,SUM/S\n");
}

```

```

else if(argc>=2) /* OK got at least one argument. */
{
    /* Get an audio channel at the highest priority */
    if( pIOA_1=SiezeChannel() )
    {

        mport1 = pIOA_1->ioa_Request.io_Message.mn_ReplyPort;
        pIOA_1->ioa_Data = playbuffer1;

        /* Get a 2nd MsgPort and 2nd IOAudio structure for double-buffering */
        pIOA_2 = AllocMem(sizeof(struct IOAudio),MEMF_PUBLIC | MEMF_CLEAR );
        mport2 = CreatePort(0,0);

        if( pIOA_2 && mport2 )
        {
            /* The 2 IOAudio requests should be initialized the same */
            /* except for the buffer and the reply port they use. */
            *pIOA_2 = *pIOA_1;
            pIOA_2->ioa_Request.io_Message.mn_ReplyPort = mport2;
            pIOA_2->ioa_Data = playbuffer2;

            /* Default is to use interleaving, not averaging */
            mainplay.USE_SUMMING = FALSE;

            /* Parse the 8SVX file and fill in the InterPlay structure */
            errorMsg = Parse8svx( argv[1] , &mainplay );

            /* If a second file name was given by the user then this is */
            /* an interleave request, so parse the 2nd 8SVX file. */
            if( argc>=3 && !errorMsg )
            {
                errorMsg = Parse8svx( argv[2] , &otherplay );
                mainplay.next_iplay = &otherplay;

                /* If the SUM keyword was given in the command line, set the */
                /* SUMMING flag so that averaging, not interleaving, is used.*/
                if( (argc == 4) &&
                    ( !strcmp(argv[3],"SUM\0") || !strcmp(argv[3],"sum\0") ) )
                {
                    mainplay.USE_SUMMING = TRUE;
                }
            }
            else
                otherplay.sample_done = 1;

            if(!errorMsg) /* File names given parsed OK? */
            {
                /* Fill up the buffer for the first request. */
                FillAudio( &mainplay, pIOA_1);

                /* Is there enough data to double-buffer ? */
                if(!mainplay.sample_done || !otherplay.sample_done)
                {
                    /* OK, enough data to double-buffer; fill up 2nd request */
                    FillAudio( &mainplay, pIOA_2 );
                    BeginIO((struct IORequest *) pIOA_1 );
                    BeginIO((struct IORequest *) pIOA_2 );

                    /* Initial state of double-buffering variables */
                    aswitch=0; pIOA=pIOA_2; mport=mport1;

```

```

/*-----*/
/*  M A I N   L O O P  */
/*-----*/

while(!mainplay.sample_done || !otherplay.sample_done)
{
    wakemask=Wait( (1 << mport->mp_SigBit) |
                  SIGBREAKF_CTRL_C );

    if( wakemask & SIGBREAKF_CTRL_C )
    {
        otherplay.sample_done = 1;
        mainplay.sample_done = 1;
    }

    while((msg=GetMsg(mport))==NULL){}

    /* Toggle double-buffering variables */
    if (aswitch) {aswitch=0;pIOA=pIOA_2;mport=mport1;}
    else         {aswitch=1;pIOA=pIOA_1;mport=mport2;}

    FillAudio( &mainplay, pIOA );
    BeginIO((struct IORequest *) pIOA );
}

wakemask=Wait( 1 << mport->mp_SigBit );
while((msg=GetMsg(mport))==NULL){}

if (aswitch) {aswitch=0;pIOA=pIOA_2;mport=mport1;}
else         {aswitch=1;pIOA=pIOA_1;mport=mport2;}

wakemask=Wait( 1 << mport->mp_SigBit );
while((msg=GetMsg(mport))==NULL){}
}
else
{
    /* Only enough data to fill up one buffer */
    BeginIO((struct IORequest *) pIOA_1 );
    wakemask=Wait( 1 << mport1->mp_SigBit );
    while((msg=GetMsg(mport1))==NULL){}
}
}
else
    /* One or the other of the files had a problem in Parse8svx() */
    printf(errorMsg);

    /* Free the memory used for the 8SVX files in Parse8svx() */
    if(mainplay.next_iplay)
        EndParse( &otherplay );
    EndParse( &mainplay );
}

else printf("Couldn't get memory for a second IOAudio and MsgPort\n");

/* Free the ports and memory used by the 2 IOAudio requests */
if(mport2) DeletePort(mport2);
if(pIOA_2) FreeMem( pIOA_2, sizeof(struct IOAudio) );

ReleaseChannel(pIOA_1);
}
else printf("Couldn't get a channel on the audio device\n");
}
else printf("Enter one or two 8SVX filenames.\n");
}

```

```

/*-----
** struct IOAudio *res = SiezeChannel( VOID )
**
** Allocates any channel at the highest priority. Once allocated,
** the hardware registers of the given channel can be hit directly
** without interfering with normal audio.device operation.
**
** Retruns NULL on failure
** or returns the address of the IOAudio used to get the channel.
** If the call to this function succeeds, ReleaseChannel() should
** be called later to free the channel and memory used for the IOAudio.
**
*/

```

```

struct IOAudio *
SiezeChannel( VOID )
{
    struct IOAudio *myAIReq=NULL;
    struct MsgPort *myAIReply=NULL;
    UBYTE chans[] = {1,2,4,8}; /* Try to get one channel, any channel */
    BYTE dev = -1;

    myAIReq=(struct IOAudio *)AllocMem(sizeof(struct IOAudio),MEMF_PUBLIC );
    if(myAIReq)
    {
        myAIReply=CreatePort(0,0);
        if(myAIReply)
        {
            myAIReq->ioa_Request.io_Message.mn_ReplyPort = myAIReply;
            myAIReq->ioa_Request.io_Message.mn_Node.ln_Pri = 127;
            myAIReq->ioa_Request.io_Command = ADCMD_ALLOCATE;
            myAIReq->ioa_AllocKey = 0;
            myAIReq->ioa_Data = chans;
            myAIReq->ioa_Length = sizeof(chans);

            dev=OpenDevice("audio.device",0L,(struct IORequest *)myAIReq,0L);

            if(! dev)
                return( myAIReq ); /* Successful exit */

            DeletePort( myAIReply );
            FreeMem( myAIReq, sizeof(struct IOAudio) );
        }
    }
    return( NULL );
}

```

```

/*-----
** VOID ReleaseChannel(struct IOAudio *rel );
**
** Frees the channel and any asociated memory allocated earlier
** with SiezeChannel().
**
*/

```

```

VOID
ReleaseChannel(struct IOAudio *rel)
{
    if(rel)
    {
        CloseDevice( (struct IORequest *) rel );
    }
}

```

```

if(rel->ioa_Request.io_Message.mn_ReplyPort)
{
    DeletePort(rel->ioa_Request.io_Message.mn_ReplyPort);
}
FreeMem( rel, sizeof(struct IOAudio) );
}

```

```

/*-----
**
** char *Parse8svx( char *filename, struct InterPlay *play_state)
**
** Pass this function the name of an 8svx file. It opens the file and
** finds the VHDR and BODY Chunks. Playback information is stored
** in the InterPlay structure.
**
** A NULL return indicates the parse was completely successful.
** A non-NULL return means the file cannot be played back for
** some reason. In that case the return value is a pointer to
** an error message explaining what went wrong.
**
** After calling Parse8svx(), End Parse() should be called
** to free any memory used.
**
*/

```

```

char *
Parse8svx(char *fname, struct InterPlay *play)
{
    BYTE iobuffer[12];
    LONG rdcount=0L;
    Chunk *pChunk=NULL;
    GroupHeader *pGH=NULL;

    Voice8Header *pV8Head = NULL;
    char *error = NULL;
    BPTR filehandle=NULL;
    BOOL NO_BODY = TRUE;
    BOOL NO_VHDR = TRUE;
}

```

```

/* Under normal operation, this function leaves the file positioned */
/* at the BODY Chunk. However, for some degenerate 8SVX files, one */
/* additional seek is needed at the end. In that case this field */
/* (play->offsetBody) will be changed to the seek offset. */
play->offsetBody = 0;
play->sample_loc = NULL; /* Set to non-NULL if memory is allocated */
play->next_iplay = NULL; /* Default is no successors, no interleave */
play->sample_done= 0L; /* Will be set to 1 when playback is done */

filehandle= NULL; /* Set to non-NULL if the file opens */

```

```

NO_BODY=TRUE;
NO_VHDR=TRUE;

```

```

/* This section just makes sure that the first 12 bytes of the */
/* file conform to the IFF FORM specification, sub-type 8SVX. */
filehandle = Open( fname, MODE_OLDFILE );
if(filehandle)
{
    /* Next, read the first 12 bytes to check the type */
}

```

```

rdcount =Read( filehandle, iobuffer, 12L );
if(rdcount==12L)
{
    /* Make sure it is an IFF FORM type */
    pGH = (GroupHeader *)iobuffer;
    if(pGH->ckID == FORM)
    {
        /* Make sure it is an 8SVX sub-type */
        if(pGH->grpSubID != ID_8SVX)
            error="Not an 8SVX file\n";
    }
    else
        error="Not an IFF FORM\n";
}
else
    error="Read error or file too short1\n";
}
else
    error="Couldn't open that file. Try another.\n";

/* Read through all Chunks until BODY and VHDR */
/* Chunks are found or until an error occurs. */
while( !error && (NO_BODY || NO_VHDR) )
{
    /* Read the first 8 bytes of the Chunk to get the type and size */
    rdcount =Read( filehandle, iobuffer, 8L );
    if(rdcount==8L)
    {
        pChunk=(Chunk *)iobuffer;
        switch(pChunk->ckID)
        {
            case ID_VHDR:
                /* AllocMem() ckSize rounded up and read */
                /* the VHDR, filling in the InterPlay */
                if(pChunk->ckSize & 1L)
                    pChunk->ckSize++;

                pV8Head = AllocMem(pChunk->ckSize, MEMF_PUBLIC);
                if(pV8Head)
                {
                    rdcount=Read(filehandle,pV8Head,pChunk->ckSize);
                    if(rdcount==pChunk->ckSize )
                    {
                        if(pV8Head->sCompression==sCmpNone)
                        {
                            /* Set the playback speed */
                            play->sample_speed = (UWORD)
                                (3579545L / pV8Head->samplesPerSec);

                            /* Set up start, end of sample data */
                            play->sample_size = pV8Head->oneShotHiSamples
                                + pV8Head->repeatHiSamples;
                        }
                        else error="Can't read compressed file\n";
                    }
                    else error="Read problem in header\n";

                    FreeMem(pV8Head, pChunk->ckSize );
                }
                else error="Couldn't get header memory\n";
                NO_VHDR = FALSE;
                break;
        }
    }
}

```

```

case ID_BODY:
    /* Technically, a VHDR could come after a BODY.*/
    /* This is a pretty unlikely occurrence though. */
    if(NO_VHDR)
    {
        if(pChunk->ckSize & 1L)
            pChunk->ckSize++;

        rdcount = Seek(filehandle, pChunk->ckSize, OFFSET_CURRENT);
        if(rdcount==-1)
            error="Problem during BODY-skipping seek\n";
        else
            play->offsetBody=rdcount;
    }
    NO_BODY = FALSE;
    break;

default:
    /* Ignore other Chunks, skipping over them */
    if(pChunk->ckSize & 1L)
        pChunk->ckSize++;

    rdcount = Seek(filehandle, pChunk->ckSize, OFFSET_CURRENT);
    if(rdcount==-1)
        error="Problem during chunk-skipping seek\n";
    break;
}
}
else error = "Read error or file too short2\n";
}

if(!error)
{
    /* In case the VHDR came after the BODY, seek back to the BODY */
    if(play->offsetBody)
    {
        rdcount = Seek(filehandle, play->offsetBody, OFFSET_BEGINNING);
        if(rdcount==-1)
            error="Couldn't seek to BODY\n";
    }

    /* OK now get the BODY data into a memory block */
    play->sample_loc = AllocMem( play->sample_size, MEMF_PUBLIC );
    if(play->sample_loc)
    {
        rdcount = Read(filehandle, play->sample_loc, play->sample_size);
        if(rdcount!=play->sample_size)
            error = "Error during BODY read\n";
        else
            play->sample_byte=play->sample_loc;
    }
    else
        error="Couldn't get memory for BODY Chunk\n";
}

if(filehandle)
    Close(filehandle);

return(error);
}

```

```

/*-----
**
** VOID EndParse( struct InterPlay * );
**
** This function simply frees any memory used by an earlier
** call to Parse8svx().
**
**-----
*/
VOID
EndParse( struct InterPlay *play )
{
if(play->sample_loc)
    FreeMem(play->sample_loc, play->sample_size );
}

/*-----
**
** VOID FillAudio(struct InterPlay *, struct IOAudio * );
**
** This function gets 512 bytes each from 2 BODY buffers and interleaves
** the bytes in the audio playback buffer.
**
**-----
-
*/
VOID
FillAudio(struct InterPlay *inplay, struct IOAudio *ioa )
{
struct InterPlay *play1,*play2;
ULONG remainder1,remainder2,x;
UWORD speedfac;
WORD value;

if(ioa->ioa_Request.io_Command != CMD_WRITE) /* For 1st time callers only */
{
    /* When two files are played at once, their speeds must match. Use */
    /* whichever speed is fastest. Interleaved requests also require the */
    /* speed to be doubled (period is halved). However, the period */
    /* cannot be lower than 124 or audio DMA bandwidth will be exceeded. */
    speedfac = inplay->sample_speed;

    if(inplay->next_iplay)
    {
        if(inplay->next_iplay->sample_speed < inplay->sample_speed)
            speedfac = inplay->next_iplay->sample_speed;

        if ( !(inplay->USE_SUMMING) )
            speedfac /= 2;
    }

    if(speedfac < 124)
        speedfac = 124;

    ioa->ioa_Request.io_Command = CMD_WRITE;
    ioa->ioa_Request.io_Flags = ADIOF_PERVOL;
    ioa->ioa_Volume = 63;
    ioa->ioa_Period = speedfac;
    ioa->ioa_Length = BUF_SIZE;
    ioa->ioa_Cycles = 1;
}

if(inplay->next_iplay)

```

```

{
play1=inplay;
play2=inplay->next_iplay;

remainder1 = play1->sample_size - (play1->sample_byte - play1->sample_loc);
remainder2 = play2->sample_size - (play2->sample_byte - play2->sample_loc);

if(play1->USE_SUMMING)
{
    /*
    ** AVERAGING LOGIC for playing TWO samples on ONE channel
    */
    for(x=0; x<BUF_SIZE ;x++)
    {
        value = 0;

        if( x<remainder1 )
        {
            value += *( (BYTE *) (play1->sample_byte) );
            play1->sample_byte++;
        }
        else if( x==remainder1 )
            play1->sample_done=1;

        if( x<remainder2 )
        {
            value += *( (BYTE *) (play2->sample_byte) );
            play2->sample_byte++;
        }
        else if( x==remainder2 )
            play2->sample_done=1;

        *(ioa->ioa_Data + x) = (UBYTE) (value/2);
    }
}
else
{
    /*
    ** INTERLEAVE LOGIC for playing TWO samples on ONE channel
    */

    /* If there are more bytes in the 1st sample data file, place them in */
    /* the EVEN positions in the playback buffer of this IOAudio request. */
    for(x=0; (x<BUF_SIZE) && (x<2*remainder1); x+=2 )
    {
        *(ioa->ioa_Data + x) = *(play1->sample_byte);
        play1->sample_byte++;
    }

    /* If there are no more bytes then mark the 1st sample as done */
    if(x<BUF_SIZE)
        play1->sample_done=1L;

    while(x<BUF_SIZE) /* Pad the playback buffer with zeroes. */
    {
        *(ioa->ioa_Data + x) = 0;
        x+=2;
    }

    /* If there are more bytes in the 2nd sample data file, place them in */
    /* the ODD positions in the playback buffer of this IOAudio request. */
    for(x=1; (x<BUF_SIZE) && (x<2*remainder2);x+=2)
    {
        *(ioa->ioa_Data + x) = *(play2->sample_byte);
        play2->sample_byte++;
    }
}

```

```

/* If there are no more bytes then mark the 2nd sample as done */
if(x<BUF_SIZE)
    play2->sample_done=1L;

while(x<BUF_SIZE)      /* Pad the playback buffer with zeroes. */
{
    *(ioa->ioa_Data + x) = 0;
    x+=2;
}
}
else
{
    /*
    ** REGULAR LOGIC for playing a single sample on a single channel.
    */
    remainder1= inplay->sample_size - (inplay->sample_byte-inplay->sample_loc);
    if(remainder1 > BUF_SIZE)
    {
        CopyMem(inplay->sample_byte,ioa->ioa_Data,BUF_SIZE);
        inplay->sample_byte+=BUF_SIZE;
    }
    else
    {
        CopyMem(inplay->sample_byte,ioa->ioa_Data,remainder1);
        ioa->ioa_Length=remainder1;
        inplay->sample_done=1L;
    }
}
}

```

