S3M player example for Youtube — © Joel Yliluoma

```
#include <string.h> /* for memcpy */
#include <stdio.h>  /* for printf,fopen,fread */
#include <assert.h>
#include <math.h>   /* for cos, exp */
#include <dos.h>    /* for outportb, enable, disable */
#include <conio.h>  /* for kbhit */

/* My custom 8-bit PCM audio library for Sound Blaster begins here */
/* Since you likely won’t program for DOS, you can skip this part  */
/* I’ll type it up really quick because that’s not our focus today */
static const unsigned SamplingRate = 36157u, ABufSize = 32768u, SB_Port = 0x220;
// Sampling rate of 36157 is chosen because it is something the PIT rate (1193181) is evenly divisible with
static unsigned char ABuf[ABufSize];
static volatile unsigned ABufTail = 0, ABufLen = 0;
struct NoIRQcontext { NoIRQcontext(){disable();} ~NoIRQcontext(){enable();} };
static void wait() { _asm hlt; }
typedef void (_far interrupt *IntVecType)();
static IntVecType OldI08;
static unsigned short IRQrun, IRQincmt;
static struct SBdriver
{   static void _far interrupt NewI08()
    {   unsigned bytes[2] = { 0x10, ABuf[ABufTail] }, IRQrun_bak = IRQrun;
        for(unsigned b = 0; b < 2; outportb(SB_Port+12, bytes[b++]))
            while(inportb(SB_Port+12) & 0x80) {/*busyloop*/};
        if(ABufLen > 0) { --ABufLen; if(++ABufTail==ABufSize) ABufTail=0; }
        ((IRQrun += IRQincmt) < IRQrun_bak) ? OldI08() : outportb(0x20, 0x20);
    }
    static void SetupIRQ0(IntVecType vec, unsigned run)
    {   NoIRQcontext ctx;
        ((IntVecType _far*)0)[8] = vec;
        for(unsigned n=0; n<3; ++n)
            outportb(0x40 + 3 * !n, n ? run >> n/2*8 : 0x34);
    }
    SBdriver()
    {   OldI08 = ((IntVecType _far*)0)[8];
        SetupIRQ0(NewI08,
                  IRQincmt = 0x1234DEUL / SamplingRate);
    }
    ~SBdriver() { SetupIRQ0(OldI08, 0); }
} SBdriver;
/* Please don’t forget to type this function */
static void AudioWrite(const unsigned char* Buf,
                              unsigned nbytes)
{
    unsigned take=ABufSize/2, from = 0;
    for(; nbytes - from >= ABufSize; from += take)
        AudioWrite(Buf+from, take);
    while(AbufLen + (nbytes - from) >= ABufSize) { wait(); }
    for(NoIRQcontext ctx; nbytes > from;
                            from += take, ABufLen += take)
    {   unsigned to = (ABufTail + ABufLen) % ABufSize,
                 t = ABufSize-to;
        memcpy(ABuf+to, Buf+from,
               take = t > nbytes-from ? nbytes-from : t);
}   }
/* End custom audio library.
 * It was longer than I anticipated it’d be...
 * AudioWrite() provides the DOS equivalent
 * of Linux OSS’s write(audio_fd,buf,nbytes).
 */
struct s3mfile
{
/* The S3M file header */
char name[28];          // song name
unsigned char eofchar, typ, dummy[2];
unsigned short ordnum, insnum, patnum, flags, cwtv, ffi;
char scrm[4];
unsigned char Vxx, Axx, Txx, mastervolume, uc,dp,dummy2[8];
unsigned short special;
unsigned char channelsettings[32];

// The rest of this struct is filled on demand
unsigned char orders[256], *patdata[100], *patend[100];

/* S3M instrument header */
struct s3minst // This mostly follows the structure in file
{
    unsigned char type;
    char filename[12];
    unsigned char filepos[3];
    long length, loopbegin, loopend;
    unsigned char volume, dummy, packflag, flags;
    unsigned long c4spd;
    unsigned char *sampledata; // internal, used in runtime
    double c4spd_factor;       // internal, used in runtime
    unsigned char name[28], scri[4];
} inst[99];
   /* Methods */
    void Load(const char* filename)
    {
        FILE *fp = fopen(filename, ”rb”);
        setbuf(fp, NULL);
        fread(name, 1, 0x60, fp); // Read S3M file header
       assert(memcmp(scrm, ”SCRM”, 4) == 0);// Verify valid module
        assert(ordnum <= 256);
        assert(insnum <= 99);
        assert(patnum <= 100);
        memset(orders, 255, sizeof(orders));
        fread(&orders, 1, ordnum, fp); // Read orders
        unsigned short inspointers[99];
        unsigned short patpointers[100];
        fread(&inspointers, 2, insnum, fp);
        fread(&patpointers, 2, patnum, fp);
        for(unsigned i=0; i < insnum; ++i)
        {
            s3mfile::s3minst& ins = inst[i];
            fseek(fp, inspointers[i]*16UL, SEEK_SET);
            fread(&ins, 1, 0x50, fp);
            assert(memcmp(ins.scri, ”SCRS”, 4) == 0);
          // Verify valid instrument
            if(ins.type != 1) continue; // Only PCM samples ok
            if(ins.length > 64000l) ins.length = 64000ul;
            if(ins.flags & 1)
            {
                assert(ins.loopbegin < ins.length);
                assert(ins.loopend   <= ins.length);
            }
            unsigned long smppos = ins.filepos[1]*0x10UL
                                 + ins.filepos[2]*0x1000UL
                               + ins.filepos[0]*0x100000UL;
           fseek(fp, smppos, SEEK_SET);
           ins.sampledata = new unsigned char [ins.length];
            assert(ins.sampledata != NULL);
            fread(ins.sampledata, 1, ins.length, fp);

            ins.c4spd_factor = 229079296.0 / ins.c4spd;
        }
        for(unsigned p=0; p < patnum; ++p)
        {
            unsigned short length = 2;
            fseek(fp, patpointers[p]*16UL, SEEK_SET);
            if(patpointers[p]) fread(&length, 1, 2, fp);
            patdata[p] = new unsigned char [ length ];
            assert(patdata[p] != NULL);
            fread(patdata[p], 1, length-2, fp);
            patend[p] = patdata[p] + length-2;
        }
        fclose(fp);
    }
/* This struct stores the runtime status of a channel */
struct Channel
{
    unsigned last_ins;
    unsigned base_note;
    double sample_offset;
    unsigned vib_offset;
    double live_period;
    double slideto_period;
    double stable_period;
    double live_hz;
    int volume;
    unsigned arpeggio;

    // Cache last command parameters, so e.g. D00 works
    unsigned last_volumeslide, last_pitchslide;
    unsigned last_vibrato_hi, last_vibrato_lo;
    unsigned last_tremolo_hi, last_tremolo_lo;
    unsigned last_portamento;
    unsigned last_arpeggio;

    Channel() :
        last_ins(0), base_note(255),
        sample_offset(0),
        stable_period(0), volume(0), arpeggio(0) { }
};
```

```
struct Slot
{
   unsigned channel, note, instrument, volume, command, infobyte;

   void Print()
   {
      static const char notenames[] = "C-C#D-D#E-F-F#G-G#A-A#B-12131415";
      printf("c%02u ", channel);
      if(note == 255) printf("...");
      else if(note == 254) printf("^^^");
      else printf("%.2s%u", notenames+2*(note&15), note>>4);
      if(instrument) printf(" %02u", instrument); else printf(" ..");
      if(volume==255) printf(" .."); else printf(" %02u", volume);
      if(command) printf(" %c", command+64); else printf(" .");
      printf("%02X", infobyte);
      fflush(stdout);
   }
};

/* Unpack a slot from packed S3M pattern data */
static Slot ReadSlot(const unsigned char*& p)
{
   unsigned char byte = *p++;
   Slot result = {byte & 0x1F, 255,0, 255, 0,0};
   if(byte & 0x20)
      { result.note = *p++; result.instrument = *p++; }
   if(byte & 0x40)
      { result.volume = *p++; }
   if(byte & 0x80)
      { result.command = *p++; result.infobyte = *p++; }
   return result;
}

/* Load a PCM sample from an instrument */
/* Update the read position as appropriate */
static double LoadSample(const s3minst& ins, double& s,
                         double inc_rate)
{
   if(s >= ins.length) return 0; // End of note

   double sample_value =
       ins.sampledata[ (unsigned) s ] - 128.0;

   s += inc_rate;
   if((ins.flags & 1) && s >= ins.loopend) /* loop? */
      s = ins.loopbegin +
          fmod(s - ins.loopbegin,
               ins.loopend - ins.loopbegin);
   return sample_value;
}

/* Play the song. */
void Play()
{
   Channel channel[32];

   double ArpeggioInterval = SamplingRate / 50.0;
    // ST3 specifies arpeggio as pitch changing 50 times in a second. Later trackers changed this to once per frame.
   double FrameDuration     = 2.5*SamplingRate / Txx;
   // Each row is 2.5 * Axx / Txx seconds long. I.e. 2.5*SamplingRate * Axx / Txx samples long.
   double HertzRatio        = 14317056.0 / SamplingRate;
   double VolumeNormalizer = (mastervolume & 127) * Vxx / 1048576.0;
   double CurrentTime      = 0;

   double NoteHzTable[16], InverseNoteHzTable[195];
   // The 195 on the above line = 15*12+15, maximum number of notes (actually 15*12+11, but incl. bad data safety.)
   {for(unsigned a=0; a<195; ++a) /* Table of 1 / 2^(x/12) */
      InverseNoteHzTable[a] = exp(a * -.0577622650466621); }
   /* pow(x,y) = exp(log(x) * y)        This is a mathematical fact. (When x > 0)
    * pow(x,-y) = 1 / pow(x,y)          This too
    * pow(2,a/-12) = exp(log(2)*a/-12). The magic number above is log(2)/-12.
    */
   {for(unsigned a=0; a<16; ++a)
      NoteHzTable[a] = exp(a * .0577622650466621); }
   // Red text, in this document, pertains to effects support (pitch slides etc.) and will be added last.

   unsigned row = 0, next_order = 0;
   while (!kbhit())
   {
      /* Load the pattern pointer */
      if(next_order >= ordnum) { next_order=0; continue; }
      unsigned current_pattern = orders[next_order++];
      if(current_pattern == 255) { next_order=0; continue; }
      if(current_pattern == 254) { continue; }
      const unsigned char* patptr = patdata[ current_pattern ];
      assert(patptr != NULL);

      /* Skip to the current row */
      printf("Skipping to row %u...¥n", row);
      for(unsigned rowskip=0; rowskip<row; ++rowskip, ++patptr)
         while( *patptr != 0) ReadSlot(patptr);

      const unsigned char* patloop = patptr;
      unsigned loops_remain      = 0;
      unsigned loop_row          = 0;

      /* Play the pattern until its end */
      int row_finish_style = 0;
      while(patptr < patend[current_pattern])
      {
         unsigned row_repeat_count = 0;

         printf("Pattern %u, row %u, remain %u...¥n", current_pattern, row, loops_remain);
         fflush(stdout);

         /* Parse the current row (repeat Axx frames, SEx times) */
         for(unsigned repeat = 0; repeat <= row_repeat_count; ++repeat)
            for(unsigned frame = 0; frame < Axx; ++frame)
            {
               const unsigned char* rowptr = patptr;

               while( *rowptr != 0)
               {
```

```
/* Parse the current slot */
Slot slot = ReadSlot(rowptr);
//if(1) { slot.Print(); printf("¥n"); }

Channel& ch = channel[slot.channel];

unsigned noteon_frame = 0, notecut_frame = 999;
unsigned sample_offset = 0;
unsigned volumeslide=0, pitchslide=0;
unsigned vibrato=0, tremolo=0;
unsigned portamento=0, arpeggio=0;

#define SetValue(name) ¥
   if(!slot.infobyte) name = ch.last_##name; ¥
   else name = ch.last_##name = slot.infobyte;

switch(slot.command) // Parse the command
{
```

```
case ’A’-64:// Axx: Set speed xx (number of frames per row)
   Axx = slot.infobyte;
   break;
case ’B’-64: // Bxx: Break to order xx row 0
   row_finish_style = 2; row = 0;
   next_order = slot.infobyte;
   break;
case ’C’-64: // Cxx: Cut to next pattern row xx
   row_finish_style = 2;
   row = (slot.infobyte >> 4)*10 + (slot.infobyte & 15);
   break;
case ’T’-64: // Txx: Set tempo xx
   Txx = slot.infobyte;
   FrameDuration = 2.5*SamplingRate / Txx;
   break;
case ’V’-64: // Vxx: Set global volume xx
   Vxx = slot.infobyte;
   VolumeNormalizer =
      (mastervolume&127) * Vxx / 1048576.0;
   break;
case ’U’-64: // Uxy: Fine pitch vibrato
   slot.infobyte |= 0x100; // passthru to Hxy
case ’H’-64: // Hxy: Pitch vibrato (4x)
   if(slot.infobyte & 0x0F)
      ch.last_vibrato_lo = slot.infobyte & 0x0F;
   else
      slot.infobyte |= ch.last_vibrato_lo;
   if(slot.infobyte & 0xF0)
      ch.last_vibrato_hi = slot.infobyte & 0xF0;
   else
      slot.infobyte |= ch.last_vibrato_hi;
   vibrato = slot.infobyte;
   break;
case ’R’-64: // Rxy: Volume vibrato
   if(slot.infobyte & 0x0F)
      ch.last_tremolo_lo = slot.infobyte & 0x0F;
   else
      slot.infobyte |= ch.last_tremolo_lo;
   if(slot.infobyte & 0xF0)
      ch.last_tremolo_hi = slot.infobyte & 0xF0;
   else
      slot.infobyte |= ch.last_tremolo_hi;
   tremolo = slot.infobyte;
   break;
case ’O’-64: // Oxx: Set sampledata offset
   sample_offset = slot.infobyte * 0x100;
   break;
```

```
case ’G’-64: // Gxx: Tone portamento
   SetValue(portamento);
   break;
case ’K’-64: // Kxy: H00 + Dxy
   vibrato = ch.last_vibrato_lo | ch.last_vibrato_hi;
   goto Dcommand;
case ’L’-64: // Lxy: G00 + Dxy
   portamento = ch.last_portamento;
   goto Dcommand;
case ’D’-64: Dcommand: // Dxy: Volume slides
   SetValue(volumeslide);
   break;
case ’E’-64: // Exy: Pitch slides down
   SetValue(pitchslide);
   break;
case ’F’-64: // Fxy: Pitch slides up
   SetValue(pitchslide);
   pitchslide |= 0x100;
   break;
case ’J’-64: // Jxy: Arpeggio 50Hz { n, n+x, n+y }
   SetValue(arpeggio);
   ch.arpeggio = arpeggio;
   break;
case ’S’-64:
   if(frame != 0) break;
   switch(slot.infobyte & 0xF0)
   {
      case 0xB0: // SBx: Pattern loop
         if(slot.infobyte == 0xB0)
           { patloop = patptr; loop_row = row; }
         else if(loops_remain == 0)
           { row_finish_style = 1;
             loops_remain = slot.infobyte & 0x0F; }
         else if(--loops_remain > 0)
             row_finish_style = 1;
         break;
      case 0xC0: // SCx: Note cut @ frame x
         notecut_frame = slot.infobyte&0x0F;
         break;
      case 0xD0: // SDx: Note delay @ frame x
         noteon_frame = slot.infobyte&0x0F;
         break;
      case 0xE0: // SEx: Row repeat x times
         row_repeat_count = slot.infobyte&0x0F;
         break;
   }
```

```
}
if(frame == noteon_frame)
{   if(slot.note == 254)
       notecut_frame = frame;
    else if(slot.note != 255 || slot.instrument != 0)
    {
       if(slot.note != 255) ch.base_note = (slot.note>>4)*12 + (slot.note&0x0F);
       if(slot.instrument) ch.last_ins = slot.instrument-1;
       if(slot.instrument && slot.volume==255)
          ch.volume = inst[ch.last_ins].volume;

       // period = 8363*16*1712 / 2^(note/12) / c4spd
       // = 229079296 * exp(note*log(2)/-12) / c4spd
       ch.slideto_period =
          InverseNoteHzTable[ch.base_note]
          * inst[ch.last_ins].c4spd_factor;
       if(!portamento)
       {
          ch.stable_period = ch.slideto_period;
          ch.sample_offset = ~~0~~ sample_offset;
       }
    }
    if(slot.volume != 255) ch.volume = slot.volume;
}
if(frame == notecut_frame) ch.base_note = 255;
```

```
if (portamento) /* pitch slide */
{   if(ch.stable_period < ch.slideto_period)
     { if ((ch.stable_period += portamento*4) >= ch.slideto_period)
            ch.stable_period = ch.slideto_period; }
    else if(ch.stable_period > ch.slideto_period)
      { if ((ch.stable_period -= portamento*4) <= ch.slideto_period)
            ch.stable_period = ch.slideto_period; }
}

ch.live_period = ch.stable_period;

if (vibrato)
{
   ch.vib_offset += vibrato >> 4;
   unsigned vib_strength = vibrato & 0x0F;
   if(vibrato < 0x100) vib_strength *= 4;
   double vib_skew = cos(ch.vib_offset * .0491);
   vib_skew *= vib_strength;
   ch.live_period = ch.stable_period + vib_skew;
   /* Note: This is inaccurate and possibly wrong. */
}
if (pitchslide)
{
   int magnitude = pitchslide & 0xFF, fine = 0;
   if(magnitude >= 0xF0) { fine=2; magnitude &= 15; }
   if(magnitude >= 0xE0) { fine=1; magnitude &= 15; }
   if(fine < 2) magnitude *= 4;
   if(pitchslide & 0x100) magnitude = -magnitude;
   if(!fine || frame == noteon_frame)
      ch.live_period = ch.stable_period += magnitude;
}
```

```
if (volumeslide)
{
   int trig = (cwtv == 0x1300 || (flags & 0x40))
           || frame >= noteon_frame
   // Fast slides trig on each frame; normal ones on all but the first.
   if((volumeslide & 0xF0) == 0x00) // D0y: slide down
      { if(trig) ch.volume -= volumeslide & 0x0F; }
   else if((volumeslide & 0x0F) == 0x00) // Dx0: slide up
      { if(trig) ch.volume += volumeslide >> 4; }
   else if(frame == noteon_frame) // Fine slides (first frame only)
   {
      if((volumeslide & 0x0F) == 0x0F) // DxF: slide up
         ch.volume += volumeslide >> 4;
      else if((volumeslide & 0xF0) == 0xF0) // DFy: slide down
         ch.volume -= volumeslide & 0x0F;
   }
   if(ch.volume < 0) ch.volume = 0;
   if(ch.volume > 64) ch.volume = 64;
}
```

```
                    if(ch.live_period != 0.0)
                        ch.live_hz = HertzRatio / ch.live_period;
                } /* End of loop for *rowptr */

                /* Mix this frame. */
                double FrameEndsAt = CurrentTime + FrameDuration;
                unsigned n_samples_to_mix = FrameEndsAt - CurrentTime;
                static unsigned char MixBuffer[1024];
                static unsigned MixBufferPos = 0;
                for(unsigned n=0; n<n_samples_to_mix; ++n)
                {
                    double result = 0.0;
                    for(unsigned c = 0; c < 32; ++c) // Each channel
                    {
                        Channel& ch = channel[c];
                        if(ch.base_note == 255) continue; // Channel is idle

                        const s3minst& ins = inst[ch.last_ins];
                        if(ch.sample_offset >= ins.length) { ch.base_note = 255; continue; }

                        double hz = ch.live_hz;
                        if(ch.arpeggio)
                        {   unsigned pos = fmod(CurrentTime/ArpeggioInterval, 3.0);
                            hz *= NoteHzTable[ (ch.arpeggio >> (pos*4)) & 15 ];
                        }
                        // At each mixing step, sample reading position is incremented
                        // by Hertz / PlaybackRate. The value for Hertz is specified
                        // in the ST3 documentation as 14317056 / Period,
                        // where Period for note N = 8363*16*1712 / (c4spd * 2^(N/12))
                        // c4spd is the middle-C finetuning value for the instrument.
                        // Note 0=C0, 1=C#0, 2=D0...; 12=C1...; 22=A#1, 23=B1, 24=C2...
                        // The Period is altered by pitch slides and vibratos.
                        result += ch.volume * LoadSample(ins, ch.sample_offset, hz);
                    }
                    int value = result * VolumeNormalizer;
                    if(value < -128) value = 128;
                    if(value > 127) value = 127;
                    MixBuffer[MixBufferPos++] = value + 128;
                    if(MixBufferPos >= 1024)
                    {   AudioWrite(MixBuffer, MixBufferPos);
                        MixBufferPos = 0;
                    }
                    CurrentTime += 1.0;
            }   } /* End of loops for Axx, SEx */
            /* Go to next row */
            if(row_finish_style == 1) { patptr = patloop; row = loop_row; continue; } // Loop
            if(row_finish_style == 2) { break; } // Jump to a different pattern
            while( *patptr != 0) ReadSlot(patptr); // This row is done, goto next
            ++patptr; ++row;
        } /* End of loop for patptr < pattern's end */
        if(row_finish_style == 0) row = 0;
    }   } /* End of loop while !kbhit, and end of Play() */
};

s3mfile s3m;
int main()
{
    s3m.Load("turbulen.s3m");
    s3m.Play();
    return 0;
}
```

The End — Written in March 2010
P.S. The colors in this code are hints for me in directing the Youtube video. – Joel Yliluoma
Exception: Comments in this color were added later for the benefit of the reader of this document
Tahoma bold = bugs I fixed after publication