

Sound Playback and Recording

Using Auto Init DMA Mode in 8 or 16 bits

Intro

This tutorial is basically an extension of the final tutorial dealing with programming the Sound Blaster card itself (#6) and the realtime mixing schemes. Pay close attention! We renamed some of our routines and stuck in a couple of nice programming tricks like function pointers to make a sweet working system! Without further hesitation, let's get into how this all fits together! Here's a peak at the header file additions/changes:

```
#define DSP_HALT_16_AUTO_INIT      0xD9
#define lobyte(x)(char)(x&0x00ff)
#define hbyte(x)(char)((x&0xff00)>>8)
Sample_ptr SampleHead;
Sample_ptr SampleTail;
Sample_ptr CurrentSample;
Sample_ptr FrontLink;
Sample_ptr EndLink;
void Play(unsigned int);
void FillBuffer8();
void FillBuffer16();
long MixSamples();
char ClipChar(long);
short ClipShort(long);
short GetSample8(Sample_ptr);
short GetSample16(Sample_ptr);
short (SB16::*GetSample)(Sample_ptr);
void (SB16::*MixingFunction)();
void Auto8MonoRecord();
void Auto8StereoRecord();
void Auto16MonoRecord();
void Auto16StereoRecord();
void SetRecordState(char);
void SetPtrFunctions();
char Recording;
```

This is the Realtime Mixing Structures tutorial code with some adjustments. Notice the use of 8 and 16 in function names. I hope this makes things easier for us! Let's start going through the code top to bottom!

```

SB16::SB16()
{ DSPVersionNum=0;
  Done=0;
  SideToFill=1;           //since starting routine will gen interrupt
  TransferLength=BUFSIZE; //by default
  SoundRate=11025;       //by default
  ModeBits=8;           //by default, later we will load from a config file!
  ModeStereoMono=0;     //Mono by default
  TransferMode=0;       //Direct Mode by default 1=ss 2=ai
  Recording=0; //Playback by default
  GetBlasterID();
  DSP_Reset();
  DisplayEnv();
  ReserveOldMixerSettings();
  Sounds= new SoundInfo[NumberOfSounds];
  WriteDSP(0xD1);       //turn speaker on if its off
  SampleHead = new SampleHeader();
  if(SampleHead == NULL)
  { cout<<"SampleHead NOT Allocated!\n";exit(1);
  }
  SampleTail = SampleHead;
}

```

So far we only introduce the Recording variable which will be set to 1 if we are recording, and 0 if we are going to do playback.

```

SB16::~~SB16()
{NewMixerSettings=OldMixerSettings;
  if(TransferMode==2) //AutoInit
  { if(ModeBits==8)
    { WriteDSP(DSP_HALT_8_AUTO_INIT);
    }
    else
    {WriteDSP(DSP_HALT_16_AUTO_INIT);
    }
  }

  SetMixerSettings();
  for(int x=0;x<NumberOfSounds;x++)
  {Unload_Sound((unsigned char*)Sounds[x].Sound);
  }
}

```

Here, we check to see if we are in AutoInitialization Mode, and if so, 8 or 16 bit mode. Send the DSP the appropriate DSP_HALT_X_AUTO_INIT to stop any current processes.

```

void SB16::SetupDMA()
{ short tsize=BUFSIZE;
  disable();
if(ModeBits==8)
  { dma.SetDMAChannel(BLASTER.DMAChan);
  }
if(ModeBits==16)
  { dma.SetDMAChannel(BLASTER.HDMAChan);
  }
dma.DisableChannel(); //Disable DMA channel while programming it
if(TransferMode==1)
  {dma.SetControlByteMask(DemandMode,AddressIncrement,SingleCycle,ReadTransfer);
  //Put SingleCycle mode
  }
else if(TransferMode==2)
  {if(Recording)
  {dma.SetControlByteMask(DemandMode,AddressIncrement,AutoInit,WriteTransfer);
  }
  else
  {dma.SetControlByteMask(DemandMode,AddressIncrement,AutoInit,ReadTransfer);
  }
  TransferLength=HALFBUFSIZE; //for SB!!!
if(ModeBits==16)
  {TransferLength>>=1;
  }
  }
dma.SetControlByte();
dma.ClearFlipFlop(); //Clear Flip-Flop
dma.SetBufferInfo();
if(ModeBits==16)
  { tsize>>=1;
  }
  dma.SetTransferLength(tsize);

enable(); //enable interrupts
dma.EnableChannel(); //enable DMA channel
}

```

This has a couple of minor changes that make a LARGE difference! We first set the dma channel according to how many bits we are using, 8 bit use the normal DMA channel, but if 16 bit, use the high DMA channel. When it comes time to call SetControlByteMask, notice the parameters in bold. These have been changed. Not only did we add an if statement to control if we are writing to or reading from memory (Recording vs. Playback), but we also changed the statement for playback. It used to say WriteTransfer. This DID work since oddly, I had assigned that what ReadTransfer is now! That's what made discovering this horrible error a good thing! Anyway, if we are using 16 bits we have to tell the DMA to transfer half as much as we really want since it will be working with 16 bits. For the same reason we tell the Sound Blaster the same thing with the TransferLength variable. Let's move on!

```

void SB16::ServiceAI()
{ char temp;
  outp(MixerAddr,0x82);
  MixingFunction();
  temp=inp(MixerData);
  if(temp & 1)
  { inp(DSPStatus);
  }
  else
  { if(temp & 2) //is the interrupt for 16 bit?
    { inp(DSPIntAck); //read in from the port
    }
  }
}

```

This function has been totally changed, although the functioning is still the same. When we get an interrupt from the Sound Blaster, we need to know who's yelling. The Sound Blaster uses 1 interrupt for 4 different processes. 8 Bit DSP, 16 Bit DSP, MPU401 and MIDI. In order to be able to do MIDI with sound playback, we need to figure out which section is requesting the interrupt and act accordingly. To figure this out, we write a 0x82 to the mixer address port, and then read a byte from the mixer data port. The bits in that char will be set to who's calling the interrupt. Bit 1 if the 8 bit Digitized I/O or MIDI is yelling, 2 if the 16 bit Digitized I/O is yelling and bit 3 if it is for the MPU401. Inside each we do the usual acknowledging the interrupt. Also notice that we are calling MixingFunction. This is one of our function pointers. It will be pointing to either the 8 or 16 bit version of our original 8 bit algorithm.

```

void SB16::SetupAutoInitDSP()
{
  if(ModeBits==8)
  { if(!ModeStereoMono)
    { if(Recording) //8 bit Mono Record
      { Auto8MonoRecord();
      }
    else
      { Auto8MonoPlay(); //8 bit Mono Playback
      }
    }
  else
  { if(Recording) //8 bit Stereo Record
    { Auto8StereoRecord();
    }
    else
      { Auto8StereoPlay(); //8 bit Stereo Playback
      }
    }
  }
  else
  { if(!ModeStereoMono)

```

```

    { if(Recording) //16 bit Mono Record
      {Auto16MonoRecord();
      }
    else
      {Auto16MonoPlay(); //16 bit Mono Playback
      }
    }
  else
    { if(Recording) //16 bit Stereo Record
      {Auto16StereoRecord();
      }
    else
      {Auto16StereoPlay();//16 bit Stereo Playback
      }
    }
  }
}

```

This function has simply been expanded to reflect the possibility that we are trying to record in any of the usual modes. Since we already have covered the playback abilities, lets cover the steps to programming the DSP for the recording. Note that we are writing a 0x42 in each of these functions. We also added the WriteDSP(0x41); for all of the playback functions at the very beginning. This used to be sent with the SetFrequency function, but I moved it outside it to increase modularity. Each of the following functions has been formatted to work with DSP version 4.xx +.

```

void SB16::Auto8MonoRecord()
{ WriteDSP(0x42); //select record
  SendFrequency();
  WriteDSP(0xCE);
  WriteDSP(0x00);
  SendLength();
}
void SB16::Auto8StereoRecord()
{ WriteDSP(0x42);
  SendFrequency();
  WriteDSP(0xCE);
  WriteDSP(0x20);
  SendLength();
}

void SB16::Auto16MonoRecord()
{ WriteDSP(0x42);
  SendFrequency();
  WriteDSP(0xBE);
  WriteDSP(0x10);
  SendLength();
}

```

```

void SB16::Auto16StereoRecord()
{ WriteDSP(0x42);
  SendFrequency();
  WriteDSP(0xBE);
  WriteDSP(0x30);
  SendLength();
}

void SB16::SetPtrFunctions()
{ if(ModeBits==16)
  { MixingFunction=FillBuffer16;
    GetSample=GetSample16;
  }
  else
  { MixingFunction=FillBuffer8; //default 8 bit mixing scheme
    GetSample=GetSample8;
  }
}

void SB16::SetRecordState(char flag)
{ Recording=flag;
}

void SB16::SetStereoMono(char stereomono)
{ ModeStereoMono=stereomono
}

void SB16::SetFrequency(long frequency)
{ SoundRate=frequency;
}

```

This function actually sets our two function pointers to valid functions. Without setting them, we will totally crash the system, TRUST ME :) We also add an access function to the Recording flag, and to two other functions which should have had them before.

```

void SB16::FillBuffer16()
{ char *start=(char*)dma.MK_FP(dma.SegInfo.rm_segment,0);
  long sample=0,i;
  SideToFill^=1;
  if(SideToFill)
  { start+=HALFBUFFSIZE;
  }
  for(i=0;i<(HALFBUFFSIZE>>1);i++,start+=2)
  { sample=ClipShort(MixSamples());
    *start= lobyte(sample);
    *(start+1)=hibyte(sample);
  }
}

```

If we are in 16 bit mode, MixingFunction will be set to this function. The only big difference is in the for loop. We traverse through our DMA buffer 2 bytes at a time. The DSP wants the low byte, followed by the high byte of the 16 bit sample. The sample is returned by MixSamples() which I should add has been changed to always return a short. I think that about covers this one!

```
short SB16::GetSample16(Sample_ptr S)
{ short temp=0;
  temp=(Sounds[S->SoundNumber].Sound[S->Position+1]<<8);
  temp+=Sounds[S->SoundNumber].Sound[S->Position];
  S->Position+=2;
  return temp;
}
```

When we need to fetch a sample from our 16 bit sound, we will get 16 bits at a time, Buuuut since we read EVERY sample 8 bits at a time (i.e. store it in a unsigned char array) we have to go through some special steps to construct the entire 16 bits from 2 separate 8 bit values. We simply take the second sample, the high bits, shift them up 8 bits and add the lower 8 bits. We then return our newly aligned 16 bit value!

```
short SB16::ClipShort(long num)
{ if(num > 32767)
  { num = 32767;
  }
  if(num < -32768)
  { num = -32768;
  }

  return (short)num;
}
```

We're going to need this little clipping function to keep our mixed samples within the data type!

Possible Implementation

```
ISRS *isrs = new ISRS();
isrs->GetSBIRQ(sb16.BLASTER.SBIntr);
sb16.GetISRPtr(isrs);
sb16.SetModeBits(16);
sb16.SetTransferMode(2);
sb16.SetStereoMono(2);
sb16.SetFrequency(44100);
sb16.SetPtrFunctions();
sb16.NewMixerSettings.vocleft=26;
sb16.NewMixerSettings.vocright=26;
sb16.NewMixerSettings.masterright=26;
sb16.NewMixerSettings.masterleft=26;
```

```
sb16.NewMixerSettings.cdleft=0;
sb16.NewMixerSettings.cdright=0;
sb16.SetMixerSettings();
sb16.Load_Sounds();
isrs->SetupSBISR();
sb16.SetupDMA();
sb16.SetupDSP();
```

We've made some subtle changes in the code and dramatically increased its usability. Now we load up our sample no matter WHAT type it is 8 or 16, stereo mono and any frequency from 44k - 5k (DSP v4.x+) and just go with it! We are also mixing these samples in real-time with the same efficiency, and with the use of function pointers, we are not checking to see what mode we are in every loop iteration so we are really saving some time!

Thanks for trying to read this. I realize that some of these changes will be hard to understand. Now that the code has been developed up to this point, we can proceed ahead unbound and ready to get into some nice, well deserved special effects! Take care! If you have any questions, comments, or some tips on how i can improve this page, please send me some Feedback!

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : deltener@mindtremors.com

Webpage : <http://www.inversereality.org>

Created by
Justin Deltener