DMA Programming

Intro

DMA or Direct Memory Access gives us the ability to move large amounts of memory from one place to another VERY quickly. It takes some CPU usage to set up the DMA tranfer, but after that, the DMA and the recipient finish without involving the CPU! This is what makes sound playback at 44khz a reality. Without DMA we'd be struggling to do anything while the sound was playing! Any computer will have 2 DMAC or 8237 DMA Controllers in which 1 is used for 8 bit transfers and the other for 16 bit transfers. If you are reading (or attempting to read) this tutorial, I'm going to assume that you are familiar with writing to ports, if you're not this might be a little confusing (if not impossible). Since we know that a normal computer has 2 DMAC's lets take a look at which ports we will be using to communicate with them!

DMAC Ports

Controller	I/O Address	Channel#	Function
DMA 1 8-bit Slave	0x00	0	Address Port
	0x01	0	Count Port
	0x02	1	Address Port
	0x03	1	Count Port
	0x04	2	Address Port
	0x05	2	Count Port
	0x06	3	Address Port
	0x07	3	Count Port
DMA 2 16-bit Master	0xC0	4	Address Port
	0xC2	4	Count Port
	0xC4	5	Address Port
	0xC6	5	Count Port
	0xC8	6	Address Port
	0xCA	6	Count Port
	0xCC	7	Address Port
	0xCE	7	Count Port

All these ports, Address and Count, what the heck does it all mean! Later on when we set up the DMA transfer we will be telling the DMA where our memory is comming from, that's where the address port comes into use. We won't need to tell it to go the Sound Blaster card, the hardware will do that for us. We will also need to tell it how much to transfer, that's where the count port comes in!

To review, we have 8 DMA Channels total although we can't use all of them because some system areas like updating RAM use some of the channels. Each channel has its own address and count ports to tell it where we are going to be getting memory from and also telling it how much to send! When telling the DMAC where we are going to fetch the memory from we also need to tell it on which memory page it is located, and those are set through a differen't set of ports, let's take a gander!

Page Registers

Here's our pretty listing of the ports we have to communicate with to tell our DMA channel which page the memory we are moving resides. We need page information to be able to move 64k in one shot! Notice that DMA channel 4 is red, this is because it is unuseable. Why you ask? Because this is where the 8 bit irq is connected (cascaded) to it! Don't worry yet about calculating the page of your memory transfer, we'll do that later :) Let's move on to mode settings! What!? Did you think that each DMA transfer was the same! How dare you!

DMAC	Address	Function and Channel
8 Bit Slave	0x87	DMA Channel 0 Page
	0x83	DMA Channel 1 Page
	0x81	DMA Channel 2 Page
	0x81	DMA Channel 3 Page
16 Bit Master	0x8F	DMA Channel 4 Page
	0x8B	DMA Channel 5 Page
	0x89	DMA Channel 6 Page
	0x8A	DMA Channel 7 Page

Mode Bit Assignments

WOW, talk about a lot of stuff to look at! All we need to know is that each bit in this byte represents a choice on how we are going to have DMA transfer the memory. I don't have the HD Space, nor the information available to explain every option. We are going to choose the following options for setting up the DMA transfer to the Sound Blaster: Demand Mode, Address Increment, Single Cycle(for now), Write Transfer and finally we are going to set the DMA channel accordingly. These are the most popular options for what we are using DMA for. Now that we can set a byte to what modes we are going to use, lets discuss where we are going to be sending it!

Bits								Function
Mode Selection Bits 7:6								
0	0							Demand Mode Selected
0	1							Single Mode Selected
1	0							Block Mode Selected
1	1							Cascade Mode Selected
Address Increment/Decrement Bit 5								
		0						Address Increment Selected
		1						Address Decrement Selected
Auto-Intialization Enable Bit 4								
			0					Single Cycle DMA Mode
			1					Auto-Initialization Mode
Transfer Type Bits 3:2								
				0	0			Verify Transfer
				0	1			Write Transfer
				1	0			Read Transfer
				1	1			Illegal
Channel Selection Bits 1:0								
						0	0	Channel 0 (4)
						0	1	Channel 1 (5)
						1	0	Channel 2 (6)
						1	1	Channel 3 (7)

We will send our control byte (if you

Write Mode Register	Register	Operation	DMAC
	0x0B	Write	8 Bit DMAC
	0xD6	Write	16 Bit DMAC

will) to one of these ports. Use 0x0B if you are programming the 8 Bit DMAC and the Channel Selection Bits will represent channels 0-3. Use 0xD6 if you are programming the 16 Bit DMAC and the Channel Selection Bits will represent channels 4-7! We are just about finished going through the DMAC ports, let's quickly go over 2 other ones!

Mask Register Control Bits

Bits								Function
0	0	0	0	0				Unused, Set to 0
Set/ Clear Mask Bit 2								
					0			Clear Mask Bit (Enable Channel)
					1			Set Mask Bit (Disable Channel)
Channel Selection Bits 1:0								
						0	0	DMA Channel 0 (4)
						0	1	DMA Channel 1 (5)
						1	0	DMA Channel 2 (6)
						1	1	DMA Channel 3 (7)

This is a control byte sorta like when we set a byte for our mode settings.
All we have to do to disable or enable a DMA Channel, is to choose the bit value for bit 2 and set the correct DMA channel! As soon as we write this byte to the Mask Register, our DMA Channel will be enable/disabled.

Single Mask Register	Register	Operation	DMAC
	0x0A	Write	8 Bit DMAC
	0xD4	Write	16 Bit DMAC

Here we are going to send our control byte to port 0x0A if our DMA channel is 3-0 or use port 0xD4 if our DMA channel is 4-7! We mostly want to disable the channel just before we start re-programming it.

Clear Byte Pointer Flip-Flop

Clear Byte Pointer F-F	Register	Operation	DMAC
	0x0C	Write	8 Bit DMAC
	0xD8	Write	16 Bit DMAC

Hey! Where's our byte layout for this port..hmmm Maybe because we can write ANYTHING to it! Thats right! We have to clear the Byte Pointer Flip Flop just after we disable our DMA channel for re-programming! That's all thereis to it!

Up to this point we've discussed what ports we are going to communicate with and under what circumstances. Lets start building our DMA class and finally go through the steps to program a full DMA transfer!

The DMA Class

Finally getting to the actual code. If you're like me you probably glanced (at best) through those port listings. GO BACK!! It will really help you out if you just take 5 minutes and not just read what the ports do but read it until it makes sence! Realize that every port has a special purpose and know which onces need specially aligned bits and such. OK! Now let's get down to defining our DMA Class! Let's go over the header file to get a brief look at what we are going to be dealing with.

```
#ifndef DMA_H__BLAH
#define DMA_H__BLAH
#ifndef LoByte
#define LoByte(x)(short)(x & 0x00FF)
#endif
#ifndef HiByte
#define HiByte(x)(short)((x&0xFF00)>>8)
#endif
#ifndef uchar
#define uchar unsigned char
#endif
//Control Byte bit definitions
//Mode Selection Bits 7:6
#define DemandMode      0       //00
#define SingleMode      64      //01
#define BlockMode       128     //10
#define CascadeMode     192     //11
//Address Increment/Decrement bit 5
#define AddressDecrement 32     //1
#define AddressIncrement 0      //0
//AutoInitialization enable bit 4
#define AutoInit        16      //1
#define SingleCycle     0       //0
//Transfer Type bits 3:2
#define VerifyTransfer  0       //00
#define WriteTransfer   4       //01
#define ReadTransfer    8       //10
//Channel Bits 1:0
#define BUFFSIZE        8192
#define HALFBUFFSIZE    4096
#include <dpmi.h>
#include <go32.h>
```

header file cont.

```
class DMA
{  public:
   DMA();
   ~DMA();
   void SetControlByteMask(uchar,uchar,uchar,uchar);
   void SetControlByte();
   void SetDMAChannel(uchar);
    void SetPorts();
   void EnableChannel();
   void DisableChannel();
   void ClearFlipFlop();
   void SetTransferLength(unsigned short);
   void AllocateDMABuffer();
   void SetBufferInfo();
   void *MK_FP(unsigned long,unsigned long);

   uchar DMAChannel, ModeByte, ControlByte, ControlByteMask, *DMABuffer ;
   uchar DMAAddrPort,DMACountPort,DMAPagePort, DMAMaskReg, DMAClearReg,
            DMAModeReg ;
   short TransferLength, page ;
   _go32_dpmi_seginfo SegInfo;
   unsigned short masksave ;
   unsigned long phys ;
};
#endif
```

Ok now this might take a little explaining. Lets do the usual top to bottom! First the entire header is enclose inside a #ifdef statement to make sure that no matter what this header will be defined only once. Now remember those special Mode Register Bits that tell the DMA how to transfer the memory? I defined each option and gave it a value, you'll see what those will do later on :) We declare our BUFFSIZE and HALFBUFFSIZE variables which represent the size of our DMA buffer! We then go into our function listing and then into our variables. Here we define all ports of type unsigned char and define some variables we know we'll need later on.Notice that we really don't have a lot of function definitions. I guess you can attribute that to my keen intellect and class building know-how (pause to reflect on my greatness) Ok enough :) Lets start with our Constructor and Destructor.

```
DMA::DMA()
{ DMAChannel=100;
 ModeByte=ControlByte=ControlByteMask=
  DMAAddrPort=DMACountPort=DMAPagePort=DMAMaskReg=DMAClearReg=DMAModeReg=0;
 EightBit=1;
 AllocateDMABuffer();
 }
DMA::~DMA()
{ _go32_dpmi_free_dos_memory(&SegInfo);
}
```

```
void DMA::AllocateDMABuffer()
{ SegInfo.size=(BUFFSIZE*2)+15/16;
  _go32_dpmi_allocate_dos_memory(&SegInfo);
  phys=SegInfo.rm_segment<<4;

 if((phys>>16)!=((phys+BUFFSIZE)>>16))
 {phys+=BUFFSIZE;
  cout<<"Hit Page Division!\n";//doing page checking right here
   }
page = (long)(phys>>16);
 memset((unsigned char *)MK_FP(phys>>4,0),0,BUFFSIZE);
}
```

Here we initialize all our variables to 0 except for DMAChannel. We intialize that to 100 because 0 is a real DMA channel number. Finally we call AllocateDMABuffer(). The constuctor simply de-allocates the DOS memory we allocated with AllocateDMABuffer().

The AllocateDMABuffer certainly looks odd! In order to do DMA transfers from a buffer to a piece of hardware, the buffer must reside in lower memory and NOT pass over a 64k page! This is an absolute requirement. To make sure of both we use the _go32 function to allocate some DOS memory according to the function requirements. We attempt to allocate 2x what we need in case we do overlap a boundary we should just be able to add BUFFSIZE onto it and have a value buffer! We then go into an if statement that checks to see if our buffer overlaps the page boundary. Finally we initialize our buffer to all 0's! Next lets setting the proper port numbers!

```
void DMA::SetDMAChannel(unsigned char channel)
{ if(channel >7)
   cout<<"Invalid DMA Channel!\n";
  else
  DMAChannel=channel;
 SetPorts();
}

void DMA::SetPorts()
{switch(DMAChannel)
  { case 0:  DMAAddrPort=0x00; DMACountPort=0x01; DMAPagePort=0x87; break;
    case 1:  DMAAddrPort=0x02; DMACountPort=0x03; DMAPagePort=0x83; break;
    case 2:  DMAAddrPort=0x04; DMACountPort=0x05; DMAPagePort=0x81; break;
    case 3:  DMAAddrPort=0x06; DMACountPort=0x07; DMAPagePort=0x82; break;
    // 16 bit channels
    case 4:  DMAAddrPort=0xC0; DMACountPort=0xC2; DMAPagePort=0x8F; break;
    case 5:  DMAAddrPort=0xC4; DMACountPort=0xC6; DMAPagePort=0x8B; break;
    case 6:  DMAAddrPort=0xC8; DMACountPort=0xCA; DMAPagePort=0x89; break;
    case 7:  DMAAddrPort=0xCC; DMACountPort=0xCE; DMAPagePort=0x8A; break;
   default: cout<<"Invalid DMA Channel!\n";break;
   }
```

f*unction cont.*

```
 if(DMAChannel < 4)
  {DMAMaskReg = 0x0A;
   DMAClearReg= 0x0C;
    DMAModeReg = 0x0B;
   }
 else
  {//16 bit channel
     DMAMaskReg  = 0xD4;
     DMAClearReg = 0xD8;
     DMAModeReg  = 0xD6;
     DMAChannel-=4;
     EightBit=0;
   }
}
```

To set all the proper ports you have to call one little function with 1 argument :) I rule :) Just call SetDMAChannel with the proper channel and everything will be set! Remember that each DMA channels has its own address, count and page ports. We set those according to what DMA channel they specifically passed. Also remember that the Mask, Clear and Mode Registers are dependent upon which DMAC we are using, the 8 Bit Slave or the 16 Bit Master. Knowing if the DMA channel is under 4 is all we need to set those! As soon as the SetDMAChannel function is called, all ports are assigned correctly :) When doing a 16 bit transer, we have to subtract 4 from the DMA channel in order for us to set it up. Notice we do this AFTER we have set the appropriate ports. Next up, let's go over setting the proper mode settings.

```
void DMA::SetControlByteMask(uchar ModeSelect,uchar AIncDec,uchar AIBit,uchar TransferB)
{ ControlByteMask=(ModeSelect+AIncDec+AIBit+TransferB);
  ControlByteMask+=DMAChannel;
}

void DMA::SetControlByte()
{ ControlByte|=ControlByteMask;
  outp(DMAModeReg,ControlByte);
}
```

In order to set all those complicated bits for the Mode Settings, all we have to do is call the SetControlByteMask function and use those #defines in our header file to fill it in! So if we were going to set our normal transfer settings it would look something like this:

```
SetControlByteMask(DemandMode,AddressIncrement,SingleCycle,ReadTransfer);
```

After calling this we can call SetControlByte when we are ready to actually program the DMA channel with those settings! Please remember that before ANY functions can be called, we have to call SetDMAChannel 1st. You will notice that the SetControlByteMask function utilizes that variable, urgo it won't work correctly if SetDMAChannel isn't called! Next, let's go over the other 2 simple registers.

```
void DMA::EnableChannel()
{ unsigned char mask=0;
  mask=DMAChannel;
  outp(DMAMaskReg,mask);
}

void DMA::DisableChannel()
{unsigned char mask=0;
 mask=DMAChannel;
 mask|=4;
 outp(DMAMaskReg,mask);
}
```

Now tell me if these don't look simple! Just as they say, both are used to disable and enable the current DMA channel! Remember the only difference between the port being set on and being set off is bit 3! Next up is the function to clear the Byte Pointer Flip-Flop.

```
void DMA::ClearFlipFlop()
{ outp(DMAClearReg,0x0000);
}
```

Now it just isn't going to get any easier than this. SetPorts made sure that DMAClearReg was set to the correct port, and hey let's just right any old number, why not 0! Next, let's find out how to send our buffersize to the Count Port.

```
void DMA::SetTransferLength(unsigned short length)
{ TransferLength=length;
  outp(DMACountPort,LoByte(length-1));//Low byte of buffersize
  outp(DMACountPort,HiByte(length-1));//High byte of buffersize
}
```

To tell the DMAC how much we need to transfer we have to communicate with the proper Count Port. We know that DMACountPort has been set correctly in SetPorts(). We just have to send the low byte of the transfersize-1, then send the high byte of the transfersize-1! Now the DMAC knows how much it is going to be transfering! Now let's tell it where to get the stuff from!

```
void DMA::SetBufferInfo()
{ unsigned long offs =(short)(phys & 0xffff);
  if(!EightBit)
    { offs=(short)(phys>>1 & 0xffff);
      }
  outp(DMAPagePort,page);
  outp(DMAAddrPort,offs&0xff);
  outp(DMAAddrPort,offs>>8);
 }
```

Here we are communicating with the DMA Address and DMA Page ports. Remember that phys was set in the AllocateDMABuffer function. We first send the low byte then follow it up with the high byte of our address. Then we finally send the page to the Page Port. If we are doing a 16 bit transfer, then we have to divide the physical address by 2 or just bitshift it once to the right.

```
void * DMA::MK_FP(unsigned long seg, unsigned long ofs)
{ if(!(_crt0_startup_flags & _CRT0_FLAG_NEARPTR))
   if(!__djgpp_nearptr_enable())
     return (void*)0;
 return (void *)(seg*16+ofs+__djgpp_conventional_base);
 }
```

I use this function to create a normal unsigned char pointer that I can use in simple string functions instead of having to deal with all hassle associated with allocating memory from DOS. A hassle in my opinion anyways :) Well we've covered the ports and what they represent, we've gone over the supporting source code that actually DOES the interfacing and communicating, now lets get a sample of how we can really use these functions!

```
void SetupDMA()
{disable();
 SetDMAChannel(OURDMACHANNEL);
  DisableChannel();                    //Disable DMA channel while programming it
  SetControlByteMask(DemandMode,AddressIncrement,SingleCycle,WriteTransfer);
  SetControlByte();                    //Put into 8-bit Single Cycle mode
  ClearFlipFlop();                     //Clear Flip-Flop
  SetBufferInfo();
 SetTransferLength(BUFFSIZE);
  enable();                            //enable interrupts
  EnableChannel();                     //enable DMA channel
 }
```

Ahhh finally success! We set the DMA channel, Disable it since we are about to program it, set our mode byte mask, set the actual control byte on the DMA, clear the flip-flop, send the location of our DMA buffer, tell the DMA how much to transfer, and finally enable the channel and we are ready to ROCK! This is the fundamental sequence to program a DMA channel!! Thanks for wallowing through this tutorial, it took about 7 hours to create so I hope it was all worth it! If you have any comments, questions, rude remarks, need to pass gas whatever...give 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