How to program the DMA

by Breakpoint

Introduction

What is the DMA?

The DMA is another chip on your motherboard (usually is an Intel 8237 chip) that allows you (the programmer) to offload data transfers between I/O boards. DMA actually stands for 'Direct Memory Access'.

An example of DMA usage would be the Sound Blaster's ability to play samples in the background. The CPU sets up the sound card and the DMA. When the DMA is told to 'go', it simply shovels the data from RAM to the card. Since this is done off-CPU, the CPU can do other things while the data is being transferred.

Lastly, if you're interested in what I know about programming the DMA to do memory to memory transfers, you might want to refer to Appendix B. This section is by no means complete, and it will probably be added to in the future as I learn more about this particular type of transfer.

Allright, here's how you program the DMA chip.

DMA Basics

When you want to start a DMA transfer, you need to know three things:

- Where the memory is located (what page)
- The offset into the page
- How much you want to transfer

Since the DMA can work in both directions (memory to I/O card, and I/O card to memory), you can see how the Sound Blaster can record as well as play by using DMA.

The DMA has two restrictions which you must abide by:

- You cannot transfer more than 64K of data in one shot
- You cannot cross a page boundary

Restriction #1 is rather easy to get around. Simply transfer the first block, and when the transfer is done, send the next block.

For those of you not familiar with pages, I'll try to explain.

Picture the first 1MB region of memory in your system. It is divided into 16 pages of 64K a piece like so:

Page	Segment:Offset address
0	0000:0000 - 0000:FFFF
1	1000:0000 - 1000:FFFF

2	2000:0000 - 2000:FFFF
3	3000:0000 - 3000:FFFF
4	4000:0000 - 4000:FFFF
5	5000:0000 - 5000:FFFF
6	6000:0000 - 6000:FFFF
7	7000:0000 - 7000:FFFF
8	8000:0000 - 8000:FFFF
9	9000:0000 - 9000:FFFF
A	A000:0000 - A000:FFFF
B	B000:0000 - B000:FFFF
C	C000:0000 - C000:FFFF
D	D000:0000 - D000:FFFF
E	E000:0000 - E000:FFFF
F	F000:0000 - F000:FFFF

This might look a bit overwhelming. Not to worry if you're a C programmer, as I'm going to assume you know the C language for the examples in this text. All the code in here will compile with Turbo C 2.0.

Okay, remember the three things needed by the DMA? Look back if you need to. We can stuff this data into a structure for easy accessing:

```
typedef struct
{
    char page;
    unsigned int offset;
    unsigned int length;
} DMA_block;
```

Now, how do we find a memory pointer's page and offset? Easy. Use the following code:

```
void LoadPageAndOffset(DMA_block *blk, char *data)
{
    unsigned int temp, segment, offset;
    unsigned long foo;
    segment = FP_SEG(data);
    offset  = FP_OFF(data);
    blk->page = (segment & 0xF000) >> 12;
    temp = (segment & 0x0FFF) << 4;
    foo = offset + temp;
    if (foo > 0xFFFF)
        blk->page++;
    blk->offset = (unsigned int)foo;
}
```

Most (if not all) of you are probably thinking, "What the heck is he doing there?" I'll explain.

The FP_SEG and FP_OFF macros find the segment and the offset of the data block in memory. Since we only need the page (look back at the table above), we can take the upper 4 bits of the segment to create our page.

The rest of the code takes the segment, adds the offset, and sees if the page needs to be advanced or not. (Note that a memory region can be located at 2FFF:F000, and a single byte increase will cause the page to increase by one.)

In plain English, the page is the highest 4 bits of the absolute 20 bit address of our memory location. The offset is the lower 12 bits of the absolute 20 bit address plus our offset.

Now that we know where our data is, we need to find the length.

The DMA has a little quirk on length. The true length sent to the DMA is actually length + 1. So if you send a zero length to the DMA, it actually transfers one byte, whereas if you send 0xFFFF, it transfers 64K. I guess they made it this way because it would be pretty senseless to program the DMA to do nothing (a length of zero), and in doing it this way, it allowed a full 64K span of data to be transferred.

Now that you know what to send to the DMA, how do you actually start it? This enters us into the different DMA channels.

DMA channels

The DMA has 4 different channels to send 8-bit data. These channels are 0, 1, 2, and 3, respectively. You can use any channel you want, but if you're transferring to an I/O card, you need to use the same channel as the card. (ie: Sound Blaster uses DMA channel 1 as a default.)

There are 3 ports that are used to set the DMA channel:

- The page register
- The address (or offset) register
- The word count (or length) register

The following chart will describe each channel and it's corresponding port number:

DMA Channel	Page	Address	Count
0	87h	0h	1h
1	83h	2h	3h
2	81h	4h	5h
3	82h	6h	7h
4	8Fh	C0h	C2h
5	8Bh	C4h	C6h
6	89h	C8h	CAh
7	8Ah	CCh	CEh

(Note: Channels 4-7 are 16-bit DMA channels. See below for more info.)

Since you need to send a two-byte value to the DMA (the offset and the length are both two bytes), the DMA requests you send the low byte of data first, then the high byte. I'll give a thorough example of how this is done momentarily.

The DMA has 3 registers for controlling it's state. Here is the bitmap layout of how they are accessed:

Mask Register (0Ah):

```
    MSB                             LSB
      x   x   x   x     x   x   x   x
      -------------------   -   -----
               |            |     |     00 - Select channel 0 mask bit
               |            |     \---- 01 - Select channel 1 mask bit
               |            |           10 - Select channel 2 mask bit
               |            |           11 - Select channel 3 mask bit
               |            |
               |            \----------  0 - Clear mask bit
               |                         1 - Set mask bit
               |
               \----------------------- xx - Don't care
```

Mask Register (0Ah):

```
    MSB                             LSB
      x   x   x   x     x   x   x   x
      \---/   -   -     -----   -----
        |     |   |       |       |     00 - Channel 0 select
        |     |   |       |       \---- 01 - Channel 1 select
        |     |   |       |             10 - Channel 2 select
        |     |   |       |             11 - Channel 3 select
        |     |   |       |
        |     |   |       |             00 - Verify transfer
        |     |   |       \------------ 01 - Write transfer
        |     |   |                     10 - Read transfer
        |     |   |
        |     |   \--------------------  0 - Autoinitialized
        |     |                          1 - Non-autoinitialized
        |     |
        |     \------------------------  0 - Address increment select
        |
        |                               00 - Demand mode
        \------------------------------ 01 - Single mode
                                        10 - Block mode
                                        11 - Cascade mode
```

DMA clear selected channel (0Ch):
Outputting a zero to this port stops all DMA processes that are currently happening as selected by the mask register (0Ah).

Some of the most common modes to program the mode register are:

- 45h: Write transfer (I/O card to memory)
- 49h: Read transfer (memory to I/O card)

Both of these assume DMA channel 1 for all transfers.

Now, there's also the 16-bit DMA channels as well. These shove two bytes of data at a time. That's how the Sound Blaster 16 works as well in 16-bit mode.

Programming the DMA for 16-bits is just as easy as 8 bit transfers. The only difference is you send data to different I/O ports. The 16-bit DMA also uses 3 other control registers as well:

Mask Register (D4h):

```
    MSB                             LSB
      x   x   x   x     x   x   x   x
      -------------------   -   -----
               |            |     |     00 - Select channel 4 mask bit
               |            |     \---- 01 - Select channel 5 mask bit
               |            |           10 - Select channel 6 mask bit
               |            |           11 - Select channel 7 mask bit
```

```
              |            |
              |            \----------  0 - Clear mask bit
              |                         1 - Set mask bit
              |
              \----------------------- xx - Don't care
```

Mode Register (D6h):

```
    MSB                             LSB
      x   x   x   x     x   x   x   x
      -----   -   -     -----   -----
        |     |   |       |       |     00 - Channel 4 select
        |     |   |       |       \---- 01 - Channel 5 select
        |     |   |       |             10 - Channel 6 select
        |     |   |       |             11 - Channel 7 select
        |     |   |       |
        |     |   |       |             00 - Verify transfer
        |     |   |       \------------ 01 - Write transfer
        |     |   |                     10 - Read transfer
        |     |   |
        |     |   \--------------------  0 - Autoinitialized
        |     |                          1 - Non-autoinitialized
        |     |
        |     \------------------------  0 - Address increment select
        |
        |                               00 - Demand mode
        \------------------------------ 01 - Single mode
                                        10 - Block mode
                                        11 - Cascade mode
```

DMA clear selected channel (D8h):
Outputting a zero to this port stops all DMA processes that are currently happening as selected by the mask register (D4h).

Now that you know all of this, how do you actually use it? Here is sample code to program the DMA using our DMA_block structure we defined before.

```
/* Just helps in making things look cleaner.  :) */
typedef unsigned char   uchar;
typedef unsigned int    uint;

/* Defines for accessing the upper and lower byte of an integer. */
#define LOW_BYTE(x)         (x & 0x00FF)
#define HI_BYTE(x)          ((x & 0xFF00) >> 8)

/* Quick-access registers and ports for each DMA channel. */
uchar MaskReg[8]   = { 0x0A, 0x0A, 0x0A, 0x0A, 0xD4, 0xD4, 0xD4, 0xD4 };
uchar ModeReg[8]   = { 0x0B, 0x0B, 0x0B, 0x0B, 0xD6, 0xD6, 0xD6, 0xD6 };
uchar ClearReg[8]  = { 0x0C, 0x0C, 0x0C, 0x0C, 0xD8, 0xD8, 0xD8, 0xD8 };

uchar PagePort[8]  = { 0x87, 0x83, 0x81, 0x82, 0x8F, 0x8B, 0x89, 0x8A };
uchar AddrPort[8]  = { 0x00, 0x02, 0x04, 0x06, 0xC0, 0xC4, 0xC8, 0xCC };
uchar CountPort[8] = { 0x01, 0x03, 0x05, 0x07, 0xC2, 0xC6, 0xCA, 0xCE };

void StartDMA(uchar DMA_channel, DMA_block *blk, uchar mode)
{
    /* First, make sure our 'mode' is using the DMA channel specified. */
    mode |= DMA_channel;

    /* Don't let anyone else mess up what we're doing. */
    disable();

    /* Set up the DMA channel so we can use it.  This tells the DMA */
    /* that we're going to be using this channel.  (It's masked) */
    outportb(MaskReg[DMA_channel], 0x04 | DMA_channel);
    /* Clear any data transfers that are currently executing. */
    outportb(ClearReg[DMA_channel], 0x00);

    /* Send the specified mode to the DMA. */
    outportb(ModeReg[DMA_channel], mode);

    /* Send the offset address.  The first byte is the low base offset, the */
    /* second byte is the high offset. */
    outportb(AddrPort[DMA_channel], LOW_BYTE(blk->offset));
    outportb(AddrPort[DMA_channel], HI_BYTE(blk->offset));

    /* Send the physical page that the data lies on. */
    outportb(PagePort[DMA_channel], blk->page);

    /* Send the length of the data.  Again, low byte first. */
    outportb(CountPort[DMA_channel], LOW_BYTE(blk->length));
    outportb(CountPort[DMA_channel], HI_BYTE(blk->length));

    /* Ok, we're done.  Enable the DMA channel (clear the mask). */
    outportb(MaskReg[DMA_channel], DMA_channel);

    /* Re-enable interrupts before we leave. */
    enable();
}

void PauseDMA(uchar DMA_channel)
{
    /* All we have to do is mask the DMA channel's bit on. */
    outportb(MaskReg[DMA_channel], 0x04 | DMA_channel);
}

void UnpauseDMA(uchar DMA_channel)
{
    /* Simply clear the mask, and the DMA continues where it left off. */
    outportb(MaskReg[DMA_channel], DMA_channel);
}

void StopDMA(uchar DMA_channel)
{
    /* We need to set the mask bit for this channel, and then clear the */
    /* selected channel.  Then we can clear the mask. */
    outportb(MaskReg[DMA_channel], 0x04 | DMA_channel);

    /* Send the clear command. */
    outportb(ClearReg[DMA_channel], 0x00);

    /* And clear the mask. */
    outportb(MaskReg[DMA_channel], DMA_channel);
}

uint DMAComplete(uchar DMA_channel)
{
    /* Register variables are compiled to use registers in C, not memory. */
    register int z;

    z = CountPort[DMA_channel];
    outportb(0x0C, 0xFF);

    /* This *MUST* be coded in Assembly!  I've tried my hardest to get it */
    /* into C, and I've had no success.  :(  (Well, at least under Borland.) */
redo:
        asm {
                mov  dx,z
                in   al,dx
                mov bl,al
                in al,dx
                mov  bh,al
                in al,dx
                mov ah,al
                in al,dx
                xchg ah,al
                sub  bx,ax
                cmp  bx,40h
                jg redo
                cmp bx,0FFC0h
                jl redo
        }
    return _AX;
}
```

I think all the above functions are self explanatory except for the last one. The last function returns the number of bytes that the DMA has transferred to (or read from) the device. I really don't know how it works as it's not my code. I found it laying on my drive, and I thought it might be somewhat useful to those of you out there. You can find out when a DMA transfer is complete this way if the I/O card doesn't raise an interrupt. DMAComplete() will return -1 (or 0xFFFF) if there is no DMA in progress.

Don't forget to load the length into your DMA_block structure as well before you call StartDMA(). (When I was writing these routines, I forgot to do that myself... I was wondering why it was transferring garbage..)

Conclusion

I hope you all have caught on to how the DMA works by now. Basically it keeps a list of DMA channels that are running or not. If you need to change something in one of these channels, you mask the channel, and reprogram. When you're done, you simply clear the mask, and the DMA starts up again.

If anyone has problems getting this to work, I'll be happy to help. Send us mail at the address below, and either I or another Tank member will fix your problem(s).

Appendix A - Programming the DMA in 32-bit protected mode

Programming the DMA in 32-bit mode is a little trickier than in 16-bit mode. One restriction you have to comply with is the 1 Mb DOS barrier. Although the DMA can access memory up to the 16 Mb limit, most I/O devices can't go above the 1 Mb area. Knowing this, we simply default to living with the 1 Mb limit.

Since your data you want to transfer is probably somewhere near the end of your RAM (Watcom allocates memory top-down), you won't have to worry about not having room in the 1 Mb area.

So, how do you actually allocate a block of RAM in the 1 Mb area? Simple. Make a DPMI call -- or better yet, use the following functions to do it for you. :)

```
typedef struct
{
    unsigned int segment;
    unsigned int offset;
    unsigned int selector;
} RMptr;

RMptr getmem(int size)
{
    union REGS regs;
    struct SREGS sregs;
    RMptr foo;

    segread(&sregs);
    regs.w.ax = 0x0100;
    regs.w.bx = (size+15) >> 4;
    int386x(0x31, &regs, &regs, &sregs);

    foo.segment = regs.w.ax;
    foo.offset = 0;
    foo.selector = regs.w.dx;
    return foo;
}

void freemem(RMptr foo)
{
    union REGS regs;
    struct SREGS sregs;

    segread(&sregs);
    regs.w.ax = 0x0101;
    regs.w.dx = foo.selector;
    int386x(0x31, &regs, &regs, &sregs);
}

void rm2pmcpy(RMptr from, char *to, int length)
{
    char far *pfrom;

    pfrom = (char far *)MK_FP(from.selector, 0);
    while (length--)
        *to++ = *pfrom++;
}

void pm2rmcpy(char *from, RMptr to, int length)
{
    char far *pto;

    pto = (char far *)MK_FP(to.selector, 0);
    while (length--)
        *pto++ = *from++;
}
```

Take note on a couple of things here. First of all, the getmem() function does exactly what it says, along with freemem(). But remember, you're not tossing around a pointer anymore. It's just a data structure with a segment and an offset stored in it.

You've allocated your memory, and now you need to put something into it. You need to use pm2rmcpy() to copy protected mode memory to real mode memory. If you want to go the other way, rm2pmcpy() is there to help you.

Now we need to load the DMA_block with our information since we now have data that the DMA can access. The function is technically the same, but it just handles different variables:

```
void LoadPageAndOffset(DMA_block *blk, RMptr data)
{
    unsigned int temp, segment, offset;
    unsigned long foo;

    segment = data.segment;
    offset  = data.offset;

    blk->page = (segment & 0xF000) >> 12;
    temp = (segment & 0x0FFF) << 4;
    foo = offset + temp;
    if (foo > 0xFFFF)
        blk->page++;
    blk->offset = (unsigned int)foo;
}
```

That's about it. Since you've now loaded your DMA_block structure with the data you need, the rest of the functions should work fine without any problems. The only thing you'll need to concern yourself with is using '_enable()' instead of 'enable()', '_disable()' instead of 'disable()', and 'outp()' instead of 'outportb()'.

Appendix B - Doing memory to memory DMA transfers

All information contained in this area is mostly theory and results of tests I have done in this area. This is not a very well documented area, and it is probably even less portable from machine to machine.

Welcome to the undocumented world of memory to memory DMA transfers! This area has given me many headaches, so as a warning (and maybe preventive medicine), you might want to take an aspirin or two before proceeding. :)

I will be writing on a level of medium intelligence. You should understand the basics of DMA transfers, and at least understand 90%, if not all, of the information contained in this document (except for this area, of course). You won't find any source code here, however, I plan to release full source code once I get the DMA to transfer a full block of memory to the video card (if it's possible)...

Anyways, let's get started.

I recently set out on the task of figuring out how to transfer a single area of memory to the video screen by using DMA.

When you sit down to think about it, it really does not seem to be too difficult. You might think, 'All I need to do is use 2 DMA channels. One set to read and one set to write. My video buffer will need to be aligned onto a segment so the DMA can transfer the data without stopping.' This is a good theory, but, unfortunately, it doesn't work. I'll show you (sort of) why it doesn't work.

I originally started out with the idea that DMA channel 0 would read from my video buffer aligned on a segment, and DMA channel 1 would write to the video memory (at 0xA000).

In testing this simple idea, I wasn't suprised that nothing happened when I enabled the DMA. After playing around with some of the registers for a little bit, I opened the Undocumented DOS book and scanned the ports. Here's a snippet of what I found:

```
0008    w   DMA channel 0-3 command register
                bit 7 = 1  DACK sense active high
                      = 0  DACK sense active low
                bit 6 = 1  DREQ sense active high
                      = 0  DREQ sense active low
                bit 5 = 1  extended write selection
                      = 0  late write selection
                bit 4 = 1  rotating priority
                      = 0  fixed priority
                bit 3 = 1  compressed timing
                      = 0  normal timing
                bit 2 = 1  enable controller
                      = 0  enable memory-to-memory
```

Seeing bit 2 at port 0x08 made me realize that the DMA might possibly NOT default to being able to handle memory to memory transfers.

Again, I tried my test program, and I still wasn't suprised that nothing happened. I opened Undocumented DOS again, and found another port that I skipped over:

```
0009       DMA write request register
```

After thinking for a little, I realized that even though the DMA is enabled, the I/O card that you are usually transferring to must communicate with the bus to tell the DMA it's ready to receive data. Since we have no I/O card to say 'Go!', we need to set the DMA to 'Go!' manually.

Undocumented DOS had no bit flags defined for port 0x09, so here is what I've been able to come up with thus far:

DMA Write Request Register (09h)

```
MSB                             LSB
  x   x   x   x     x   x   x   x
  -------------------   -   -----
           |            |     |     00 - Select channel 0
           |            |     \---- 01 - Select channel 1
           |            |           10 - Select channel 2
           |            |           11 - Select channel 3
           |            |
           |            \----------  0 - ???
           |                         1 - Needs to be turned on
           |
           \----------------------- xx - Don't care
```

After adding a couple of lines of code, and running the test program once again, I was amazed to see that my screen cleared! I didn't get a buffer copy, I got a screen clear. I went back into the code to make sure my buffer had data, and sure enough, it did.

Wondering what color my screen had cleared, I added some more code and found that the screen was cleared with value 0xFF.

Pondering on this one, I made the assumption that the DMA is NOT receiving data from itself, but from the bus! Since there are no I/O cards to send data down the bus, I assumed that 0xFF was a default value.

But then again, maybe DMA channel 0 wasn't working right. I took the lines of code to initialize DMA channel 0 and the code to start the DMA transfer for channel 0 out of the code and reran the test code. Much to my suprise, the screen cleared twice as fast as before.

As for timing, my results aren't too accurate. In fact, don't even take these as being true. The first test (with both DMA 0 and 1 enabled), cranked out around 8.03 frames per second on my 486DX-33 VLB Cirrus Logic. The second test (with just DMA 1 enabled), cranked out 18.23 fps.

This is about as far as I've gotten with memory to memory DMA transfers. I'm going to be trying other DMA channels, and maybe even the 16-bit ones to get a faster dump.

If anyone can contribute any information, please let me know. You will be credited for any little tiny piece of help you can give. Maybe if we all pull together, we might actually be able to do frame dumps in the background while we're rendering our next frame... could prove to be useful!

You can reach me at 'nstalker@iag.net'.
- Breakpoint