Programação DMA

Introdução

DMA ou Direct Memory Access nos dá a capacidade de mover grandes quantidades de memória de um lugar para outro MUITO rapidamente. É preciso um pouco de uso da CPU para configurar a transferência DMA, mas depois disso, o DMA e o destinatário terminam sem envolver a CPU! É isso que torna a reprodução de som a 44khz uma realidade. Sem DMA, teríamos dificuldade para fazer qualquer coisa enquanto o som estivesse tocando! Qualquer computador terá 2 controladores DMAC ou 8237 DMA nos quais 1 é usado para transferências de 8 bits e o outro para transferências de 16 bits. Se você estiver lendo (ou tentando ler) este tutorial, vou assumir que você está familiarizado com a escrita em portas, se não estiver, isso pode ser um pouco confuso (se não impossível). Como sabemos que um computador normal tem 2 DMACs, vamos dar uma olhada em quais portas usaremos para nos comunicar com eles!

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

Todas essas portas, Endereço e Contagem, o que diabos isso tudo significa! Mais tarde, quando configurarmos a transferência DMA, diremos ao DMA de onde nossa memória está vindo, é aí que a porta de endereço entra em uso. Não precisaremos dizer a ele para ir para a placa Sound Blaster, o hardware fará isso por nós. Também precisaremos dizer a ele quanto transferir, é aí que a porta de contagem entra!

Para revisar, temos 8 Canais DMA no total, embora não possamos usar todos eles porque algumas áreas do sistema, como atualização de RAM, usam alguns dos canais. Cada canal tem seu próprio endereço e conta portas para dizer de onde vamos obter memória e também dizer quanto enviar! Ao dizer ao DMAC de onde vamos buscar a memória, também precisamos dizer em qual página de memória ela está localizada, e essas são definidas por meio de um conjunto diferente de portas, vamos dar uma olhada!

Page Registers

Aqui está nossa bela lista de portas com as quais temos que comunicar para dizer ao nosso canal DMA em qual página a memória que estamos moven do reside. Precisamos de informações de página para poder mover 64k de uma só vez! Observe que o canal DMA 4 está vermelho, isso porque ele é inutilizável. Por que você pergunta? Porque é aqui que o irq de 8 bits está conectado (em cascata) a ele! Não se preocupe ainda em calcular a página da sua transferência de memória, faremos isso mais tarde :) Vamos passar para as configurações do modo! O quê!? Você achou que cada transferência DMA era a mesma! Como você ousa!

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

Atribuições de bits de modo

UAU, fala sobre muita coisa para olhar! Tudo o que precisamos saber é que cada bit neste byte representa uma escolha sobre como vamos ter o DMAtransferindo a memória. Não tenho o Espaço em HD, nem as informações disponíveis para explicar todas as opções. Nós vamos escolher as seguintes opções para configurar a transferência de DMA para a Sound Blaster: Modo de Demanda, Incremento de Endereço, Ciclo Único (por enquanto), Transferência de Gravação e finalmente vamos definir o canal DMA de acordo. Essas são as opções mais populares para o que estamos usando o DMA. Agora que podemos definir um byte para quais modos vamos usar, vamos discutir para onde vamos enviá-lo!

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)

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

Enviaremos nosso byte de controle (se preferir) para uma dessas portas. Use 0x0B se estiver programando o DMAC de 8 bits e os bits de seleção de canal representarão os canais 0-3. Use 0xD6 se estiver programando o DMAC de 16 bits e os bits de seleção de canal representarão os canais 4-7! Estamos quase terminando de passar pelas portas DMAC, vamos rapidamente passar por outras 2!

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)

Este é um byte de controle, como quando definimos um byte para nossas configurações de modo.Tudo o que temos que fazer para desabilitar ou habilitar um Canal DMA é escolher o valor do bit para o bit 2 e definir o canal DMA correto! Assim que escrevermos este byte no Registro de Máscara, nosso Canal DMA será habilitado/desabilitado.

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

Aqui, enviaremos nosso byte de controle para a porta 0x0A se nosso canal DMA for 3-0 ou usaremos a porta 0xD4 se nosso canal DMA for 4-7! Queremos principalmente desabilitar o canal antes de começarmos a reprogramá-lo.

Clear Byte Pointer Flip-Flop

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

Ei! Onde está nosso layout de bytes para esta porta... hmmm Talvez porque podemos escrever QUALQUER COISA nela! Isso mesmo! Temos que limpar o Byte Pointer Flip Flop logo após desabilitar nosso canal DMA para reprogramação! É só isso!

Até este ponto, discutimos com quais portas iremos nos comunicar e sob quais circunstâncias. Vamos começar a construir nossa classe DMA e, finalmente, passar pelos passos para programar uma transferência DMA completa!

The DMA Class

Finalmente chegando ao código real. Se você for como eu, provavelmente deu uma olhada (na melhor das hipóteses) nessas listagens de portas. VOLTE!! Vai realmente ajudar você se você tirar apenas 5 minutos e não apenas ler o que as portas fazem, mas ler até que faça sentido! Perceba que cada porta tem um propósito especial e saiba quais precisam de bits especialmente alinhados e tal. OK! Agora vamos começar a definir nossa Classe DMA! Vamos revisar o arquivo de cabeçalho para dar uma olhada rápida no que vamos lidar.

```
#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>
```

```
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, agora isso pode exigir uma pequena explicação. Vamos fazer o de cima para baixo de sempre! Primeiro, o cabeçalho inteiro é colocado dentro de uma declaração #ifdef para garantir que, não importa o que, esse cabeçalho será definido apenas uma vez. Agora, lembra daqueles Bits de Registro de Modo especiais que dizem ao DMA como transferir a memória? Eu defini cada opção e dei a ela um valor, você verá o que eles farão mais tarde :) Declaramos nossas variáveis BUFFSIZE e HALFBUFFSIZE que representam o tamanho do nosso buffer DMA! Então, vamos para nossa listagem de funções e depois para nossas variáveis. Aqui, definimos todas as portas do tipo unsigned char e definimos algumas variáveis que sabemos que precisaremos mais tarde. Observe que realmente não temos muitas definições de função. Acho que você pode atribuir isso ao meu intelecto aguçado e conhecimento de construção de classes (pausa para refletir sobre minha grandeza) Ok, chega :) Vamos começar com nosso Construtor e Destrutor.

```
DMA::DMA()
{
    DMAChannel=100;
    ModeByte=ControlByte=ControlByteMask=

DMAAddrPort=DMACountPort=DMAPagePort=DMAMaskReg=DMAClearReg=DMAMode
Reg=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);
}
```

Aqui inicializamos todas as nossas variáveis para 0, exceto para DMAChannel. Inicializamos isso para 100 porque 0 é um número de canal DMA real. Finalmente, chamamos AllocateDMABuffer(). O construtor simplesmente desaloca a memória DOS que alocamos com AllocateDMABuffer(). O AllocateDMABuffer certamente parece estranho! Para fazer transferências DMA de um buffer para um pedaço de hardware, o buffer deve residir em memória inferior e NÃO passar por uma página de 64k! Este é um requisito absoluto. Para garantir ambos, usamos a função _go32 para alocar alguma memória DOS de acordo com os requisitos da função. Tentamos alocar 2x o que precisamos, caso sobreponhamos um limite, devemos apenas ser capazes de adicionar BUFFSIZE a ele e ter um buffer de valor! Em seguida, entramos em uma instrução if que verifica se nosso buffer sobrepõe o limite da página. Finalmente, inicializamos nosso buffer para todos os 0! Em seguida, vamos definir os números de porta adequados!

```
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;
```

```
    }

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

Para definir todas as portas adequadas, você precisa chamar uma pequena função com 1 argumento :) Eu decido :) Basta chamar SetDMAChannel com o canal adequado e tudo estará definido! Lembre-se de que cada canal DMA tem seu próprio endereço, contagem e portas de página. Definimos isso de acordo com o canal DMA que eles passaram especificamente. Lembre-se também de que os registradores Mask, Clear e Mode dependem de qual DMAC estamos usando, o Slave de 8 bits ou o Master de 16 bits. Saber se o canal DMA está abaixo de 4 é tudo o que precisamos para defini-los! Assim que a função SetDMAChannel é chamada, todas as portas são atribuídas corretamente :) Ao fazer uma transferência de 16 bits, temos que subtrair 4 do canal DMA para podermos configurá-lo. Observe que fazemos isso DEPOIS de definir as portas apropriadas. A seguir, vamos definir as configurações de modo adequadas

```
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);
}
```

Para definir todos esses bits complicados para as Configurações de Modo, tudo o que temos que fazer é chamar a função SetControlByteMask e usar essas #defines em nosso arquivo de cabeçalho para preenchê-lo! Então, se fôssemos definir nossas configurações de transferência normais, seria algo como isto: SetControlByteMask(DemandMode,AddressIncrement,SingleCycle,ReadTransfer); Depois de chamar isso, podemos chamar SetControlByte quando estivermos prontos para realmente programar o canal DMA com essas configurações! Lembre-se de que antes que QUALQUER função possa ser chamada, temos que chamar

SetDMAChannel primeiro. Você notará que a função SetControlByteMask utiliza essa variável, urgo, ela não funcionará corretamente se SetDMAChannel não for chamada! Em seguida, vamos examinar os outros 2 registradores simples.

```
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);
}
```

Agora me diga se isso não parece simples! Como dizem, ambos são usados para desabilitar e habilitar o canal DMA atual! Lembre-se de que a única diferença entre a porta ser definida como ligada e desligada é o bit 3! A próxima é a função para limpar o Byte Pointer Flip-Flop.

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

Agora não vai ficar mais fácil do que isso. SetPorts garantiu que DMAClearReg foi definido para a porta correta, e ei, vamos corrigir qualquer número antigo, por que não 0! Em seguida, vamos descobrir como enviar nosso buffersize para a 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
}
```

Para dizer ao DMAC quanto precisamos transferir, temos que nos comunicar com a Count Port adequada. Sabemos que DMACountPort foi definido corretamente em SetPorts(). Só temos que enviar o byte baixo do transfersize-1, depois enviar o byte alto do transfersize-1! Agora o DMAC sabe quanto vai transferir! Agora vamos dizer a ele de onde obter o material!

```
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);
}
```

Aqui estamos nos comunicando com as portas DMA Address e DMA Page. Lembre-se de que phys foi definido na função AllocateDMABuffer. Primeiro enviamos o byte baixo e depois o seguimos com o byte alto do nosso endereço. Então, finalmente enviamos a página para a Page Port. Se estivermos fazendo uma transferência de 16 bits, então temos que dividir o endereço físico por 2 ou apenas deslocá-lo uma vez para a direita.

```
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);
}
```

Eu uso essa função para criar um ponteiro char unsigned normal que eu possa usar em funções de string simples em vez de ter que lidar com todo o incômodo associado à alocação de memória do DOS. Um incômodo na minha opinião de qualquer maneira :) Bem, nós cobrimos as portas e o que elas representam, nós revisamos o código-fonte de suporte que realmente FAZ a interface e a comunicação, agora vamos ter uma amostra de como podemos realmente usar essas funções!

```
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 finalmente sucesso! Definimos o canal DMA, desabilitamos, pois estamos prestes a programá-lo, definimos nossa máscara de byte de modo, definimos o byte de controle real no DMA, limpamos o flip-flop, enviamos a localização do nosso buffer DMA, informamos ao DMA quanto transferir e, finalmente, habilitamos o canal e estamos prontos para o ROCK! Esta é a sequência fundamental para programar um canal DMA!! Obrigado por mergulhar neste tutorial, levou cerca de 7 horas para criar, então espero que tenha valido a pena! Se você tiver algum comentário, pergunta, comentário rude, precisar soltar gases, seja lá o que for... me dê um feedback!!

Sinta-se à vontade para me enviar um e-mail sobre qualquer coisa. Não posso garantir que serei de ALGUMA ajuda, mas com certeza tentarei :-)

Email : deltener@mindtremors.com
Webpage : http://www.invereality.org

Como programar o DMA

Introdução

O que é o DMA?

O DMA é outro chip na sua placa-mãe (geralmente é um chip Intel 8237) que permite que você (o programador) descarregue transferências de dados entre placas de E/S. DMA na verdade significa "Direct Memory Access".

Um exemplo de uso de DMA seria a capacidade da Sound Blaster de tocar samples em segundo plano. A CPU configura a placa de som e o DMA. Quando o DMA é instruído a "ir", ele simplesmente empurra os dados da RAM para a placa. Como isso é feito fora da CPU, a CPU pode fazer outras coisas enquanto os dados estão sendo transferidos.

Por fim, se você estiver interessado no que eu sei sobre programar o DMA para fazer transferências de memória para memória, você pode querer consultar o Apêndice B. Esta seção não está de forma alguma completa, e provavelmente será adicionada no futuro conforme eu aprender mais sobre esse tipo específico de transferência.

Tudo bem, aqui está como você programa o chip DMA.

Básico de DMA

Quando você deseja iniciar uma transferência DMA, você precisa saber três coisas:

- Onde a memória está localizada (qual página)
- O deslocamento na página
- Quanto você deseja transferir

Como o DMA pode funcionar em ambas as direções (memória para placa de E/S e placa de E/S para memória), você pode ver como a Sound Blaster pode gravar e reproduzir usando DMA.

O DMA tem duas restrições que você deve respeitar:

- Você não pode transferir mais de 64K de dados de uma só vez
- Você não pode cruzar um limite de página

A restrição nº 1 é bem fácil de contornar. Simplesmente transfira o primeiro bloco e, quando a transferência for concluída, envie o próximo bloco.

Para aqueles que não estão familiarizados com páginas, tentarei explicar.

Imagine a primeira região de 1 MB de memória em seu sistema. Ela é dividida em 16 páginas de 64K cada, assim:

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 Isso pode parecer um pouco assustador. Não se preocupe se você for um programador C, pois vou assumir que você conhece a linguagem C para os exemplos neste texto. Todo o código aqui será compilado com o Turbo C 2.0.

Ok, lembra das três coisas necessárias para o DMA? Olhe para trás se precisar. Podemos colocar esses dados em uma estrutura para fácil acesso:

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

Agora, como encontramos a página e o deslocamento de um ponteiro de memória? Fácil. Use o seguinte código:

```
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)  0xFFFF)
        blk->page++;
    blk->offset = (unsigned int)foo;
}
```

A maioria (se não todos) de vocês provavelmente está pensando: "O que diabos ele está fazendo lá?" Vou explicar.

As macros FP_SEG e FP_OFF encontram o segmento e o deslocamento do bloco de dados na memória. Como precisamos apenas da página (veja novamente a tabela acima), podemos pegar os 4 bits superiores do segmento para criar nossa página.

O restante do código pega o segmento, adiciona o deslocamento e vê se a página precisa ser avançada ou não. (Observe que uma região de memória pode ser localizada em 2FFF:F000, e um único aumento de byte fará com que a página aumente em um.)

Em inglês simples, a página são os 4 bits mais altos do endereço absoluto de 20 bits do nosso local de memória. O deslocamento são os 12 bits mais baixos do endereço absoluto de 20 bits mais nosso deslocamento.

Agora que sabemos onde nossos dados estão, precisamos encontrar o comprimento.

O DMA tem uma pequena peculiaridade no comprimento. O comprimento real enviado para o DMA é, na verdade, comprimento + 1. Então, se você enviar um comprimento zero para o DMA, ele realmente transfere um byte, enquanto se você enviar 0xFFFF, ele transfere 64K. Eu acho que eles fizeram dessa forma porque seria bem sem sentido programar o DMA para não fazer nada (um comprimento de zero), e ao fazer dessa forma, permitiu que um intervalo completo de 64K de dados fosse transferido.

Agora que você sabe o que enviar para o DMA, como você realmente inicia? Isso nos insere nos diferentes canais do DMA.

Canais de DMA.

O DMA tem 4 canais diferentes para enviar dados de 8 bits. Esses canais são 0, 1, 2 e 3, respectivamente. Você pode usar qualquer canal que quiser, mas se estiver transferindo para uma placa de E/S, precisará usar o mesmo canal da placa. (por exemplo: a Sound Blaster usa o canal DMA 1 como padrão.)

Há 3 portas que são usadas para definir o canal DMA:

- O registrador de página
- O registrador de endereço (ou deslocamento)
- O registrador de contagem de palavras (ou comprimento)

O gráfico a seguir descreverá cada canal e seu número de porta correspondente:

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: Canais de 4-7 são canais de 16-bit de DMA. Veja mais informações abaixo.)

Como você precisa enviar um valor de dois bytes para o DMA (o deslocamento e o comprimento são ambos dois bytes), o DMA solicita que você envie o byte baixo de dados primeiro, depois o byte alto. Darei um exemplo completo de como isso é feito momentaneamente.

O DMA tem 3 registradores para controlar seu estado. Aqui está o layout de bitmap de como eles são acessados:

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):
A saída de um zero para esta porta interrompe todos os processos DMA que estão acontecendo atualmente, conforme selecionado pelo registro de máscara (0Ah).

Alguns dos modos mais comuns para programar o registro de modo são:

• 45h: Transferência de gravação (placa de E/S para memória)

• 49h: Transferência de leitura (memória para placa de E/S)

Ambos assumem o canal DMA 1 para todas as transferências.

Agora, há também os canais DMA de 16 bits. Eles empurram dois bytes de dados por vez. É assim que a Sound Blaster 16 funciona também no modo de 16 bits.

Programar o DMA para 16 bits é tão fácil quanto transferências de 8 bits. A única diferença é que você envia dados para diferentes portas de E/S. O DMA de 16 bits também usa 3 outros registros de controle:

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):
A saída de um zero para esta porta interrompe todos os processos DMA que estão acontecendo atualmente, conforme selecionado pelo registro de máscara (D4h).

Agora que você sabe tudo isso, como você realmente usa isso? Aqui está um código de exemplo para programar o DMA usando nossa estrutura DMA_block que definimos antes.

```
/* 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;
}
```

Acho que todas as funções acima são autoexplicativas, exceto a última. A última função retorna o número de bytes que o DMA transferiu para (ou leu do) dispositivo. Realmente não sei como funciona, pois não é meu código. Encontrei-o no meu drive e pensei que poderia ser útil para vocês. Você pode descobrir quando uma transferência de DMA é concluída dessa forma se a placa de E/S não gerar uma interrupção. DMAComplete() retornará -1 (ou 0xFFFF) se não houver DMA em andamento.

Não se esqueça de carregar o comprimento na sua estrutura DMA_block também antes de chamar StartDMA(). (Quando eu estava escrevendo essas rotinas, esqueci de fazer isso sozinho... Fiquei me perguntando por que ele estava transferindo lixo..

Conclusão:

Espero que todos vocês tenham entendido como o DMA funciona até agora. Basicamente, ele mantém uma lista de canais DMA que estão em execução ou não. Se você precisar alterar algo em um desses canais, você mascara o canal e reprograma. Quando terminar, você simplesmente limpa a máscara e o DMA inicia novamente.

Se alguém tiver problemas para fazer isso funcionar, ficarei feliz em ajudar. Envie-nos um e-mail no endereço abaixo, e eu ou outro membro do Tank consertaremos seu(s) problema(s).

Apendice A - Programação do DMA em modo protegido de 32 bits

Programar o DMA no modo de 32 bits é um pouco mais complicado do que no modo de 16 bits. Uma restrição que você precisa cumprir é a barreira DOS de 1 Mb. Embora o DMA possa acessar a memória até o limite de 16 Mb, a maioria dos dispositivos de E/S não pode ir além da área de 1 Mb. Sabendo disso, simplesmente usamos o limite de 1 Mb como padrão.

Como os dados que você deseja transferir provavelmente estão em algum lugar próximo ao fim da sua RAM (o Watcom aloca a memória de cima para baixo), você não precisa se preocupar em não ter espaço na área de 1 Mb.

Então, como você realmente aloca um bloco de RAM na área de 1 Mb? Simples. Faça uma chamada DPMI -- ou melhor ainda, use as seguintes funções para fazer isso por você. :)

```
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++;
}
```

Tome nota de algumas coisas aqui. Primeiro de tudo, a função getmem() faz exatamente o que diz, junto com freemem(). Mas lembre-se, você não está mais jogando um ponteiro. É apenas uma estrutura de dados com um segmento e um deslocamento armazenado nela.

Você alocou sua memória e agora precisa colocar algo nela. Você precisa usar pm2rmcpy() para copiar a memória do modo protegido para a memória do modo real. Se você quiser ir para o outro lado, rm2pmcpy() está lá para ajudar.

Agora precisamos carregar o DMA_block com nossas informações, pois agora temos dados que o DMA pode acessar. A função é tecnicamente a mesma, mas ela apenas manipula variáveis diferentes:

```
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)  0xFFFF)
        blk->page++;
    blk->offset = (unsigned int)foo;
}
```

É mais ou menos isso. Como você carregou sua estrutura DMA_block com os dados que precisa, o resto das funções deve funcionar bem sem problemas. A única coisa com que você precisa se preocupar é usar '_enable()' em vez de 'enable()', '_disable()' em vez de 'disable()' e 'outp()' em vez de 'outportb()'.

Apendice B - Fazendo transferências DMA de memória para memória

Todas as informações contidas nesta área são principalmente teoria e resultados de testes que fiz nesta área. Esta não é uma área muito bem documentada e provavelmente é ainda menos portátil de máquina para máquina.

Bem-vindo ao mundo não documentado das transferências DMA de memória para memória! Esta área me deu muitas dores de cabeça, então, como um aviso (e talvez um remédio preventivo), você pode querer tomar uma aspirina ou duas antes de prosseguir. :)

Escreverei em um nível de inteligência média. Você deve entender os fundamentos das transferências DMA e entender pelo menos 90%, se não todas, das informações contidas neste documento (exceto por esta área, é claro). Você não encontrará nenhum código-fonte aqui, no entanto, planejo liberar o código-fonte completo assim que eu conseguir que o DMA transfira um bloco completo de memória para a placa de vídeo (se for possível)...

De qualquer forma, vamos começar.

Recentemente, comecei a tarefa de descobrir como transferir uma única área de memória para a tela de vídeo usando DMA.

Quando você se senta para pensar sobre isso, realmente não parece ser muito difícil. Você pode pensar, 'Tudo o que preciso fazer é usar 2 canais DMA. Um definido para ler e um definido para escrever. Meu buffer de vídeo precisará ser alinhado em um segmento para que o DMA possa transferir os dados sem parar.' Esta é uma boa teoria, mas, infelizmente, não funciona. Vou mostrar a você (mais ou menos) por que não funciona.

Originalmente, comecei com a ideia de que o canal DMA 0 leria do meu buffer de vídeo alinhado em um segmento, e o canal DMA 1 escreveria na memória de vídeo (em 0xA000).

Ao testar esta ideia simples, não fiquei surpreso que nada aconteceu quando habilitei o DMA. Depois de brincar um pouco com alguns dos registros, abri o livro Undocumented DOS e escaneei as portas. Aqui está um trecho do que encontrei:

```
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
```

Ver o bit 2 na porta 0x08 me fez perceber que o DMA pode NÃO ter como padrão a capacidade de manipular transferências de memória para memória.

Mais uma vez, tentei meu programa de teste e ainda não fiquei surpreso que nada aconteceu. Abri o Undocumented DOS novamente e encontrei outra porta que pulei:

```
0009        DMA write request register
```

Depois de pensar um pouco, percebi que, embora o DMA esteja habilitado, a placa de E/S para a qual você normalmente está transferindo deve se comunicar com o barramento

para informar ao DMA que está pronto para receber dados. Como não temos uma placa de E/S para dizer 'Go!', precisamos definir o DMA para 'Go!' manualmente.

O DOS não documentado não tinha sinalizadores de bits definidos para a porta 0x09, então aqui está o que consegui descobrir até agora:

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
```

Depois de adicionar algumas linhas de código e executar o programa de teste mais uma vez, fiquei surpreso ao ver que minha tela foi limpa! Não obtive uma cópia do buffer, obtive uma tela limpa. Voltei ao código para ter certeza de que meu buffer tinha dados e, com certeza, tinha.

Imaginando qual cor minha tela havia sido limpa, adicionei mais código e descobri que a tela foi limpa com o valor 0xFF.

Ponderando sobre isso, presumi que o DMA NÃO está recebendo dados de si mesmo, mas do barramento! Como não há placas de E/S para enviar dados pelo barramento, presumi que 0xFF era um valor padrão.

Mas, novamente, talvez o canal 0 do DMA não estivesse funcionando direito. Peguei as linhas de código para inicializar o canal 0 do DMA e o código para iniciar a transferência do DMA para o canal 0 do código e executei novamente o código de teste. Para minha surpresa, a tela foi limpa duas vezes mais rápido do que antes.

Quanto ao tempo, meus resultados não são muito precisos. Na verdade, nem pense nisso como verdade. O primeiro teste (com DMA 0 e 1 habilitados) produziu cerca de 8,03 quadros por segundo no meu 486DX-33 VLB Cirrus Logic. O segundo teste (com apenas DMA 1 habilitado) produziu 18,23 fps.

Isso é o máximo que consegui com transferências DMA de memória para memória. Vou tentar outros canais DMA, e talvez até os de 16 bits para obter um dump mais rápido.

Se alguém puder contribuir com alguma informação, por favor, me avise. Você será creditado por qualquer pequena ajuda que puder dar. Talvez se todos nós nos unirmos, possamos realmente fazer dumps de quadros em segundo plano enquanto renderizamos nosso próximo quadro... pode ser útil!