

```

*****
D M A U T I L . P A S
*****
{
Task      : Provides functions for programming the DMA
            controller.
}
*****
{
Author : Michael Tischer / Bruno Jennrich
Developed on : 03/20/1994
Last update : 04/5/1995
*****
}

Unit DMAUTIL;

Interface

Const
{- Bits of status register ($08, $D0) Read -----}
STATUS_REQ3 = $80; { Bit set: specific DMA channel }
STATUS_REQ2 = $40; { gets DMA request. }
STATUS_REQ1 = $20; { Request }
STATUS_REQ0 = $10;
STATUS_TC3 = $08; { Bit set: Since the last read of }
STATUS_TC2 = $04; { status register a DMA }
STATUS_TC1 = $02; { transfer has been concluded. }
STATUS_TC0 = $01; { Terminal Count }

{- Bit of command register ($08, $D0) Write =====}

COMMAND_DACKLEVEL = $80; { Bit 7 set: DMA acknowledge level }
{ HIGH active }
COMMAND_DREQLEVEL = $40; { Bit 6 set: REQ acknowledge level }
{ LOW active }
COMMAND_EXTWRITE = $20; { Bit 5 set: EXTENDED Write, }
{ else LATE Write }
COMMAND_FIXEDPRI = $10; { Bit 4 set: fixed priority }
COMMAND_COMPRESS = $08; { Bit 3 set: compressed cycling }
COMMAND_INACTIVE = $04; { Bit 2 set: controller inactive }
COMMAND_ADH0 = $02; { Bit 1 set: address hold for channel }
{ 0/4 inactive }
COMMAND_MEM2MEM = $01; {- Bit 0 set: memory/memory, }
{ else memory/periphery }

{- Bits of request registers ( $09, $D2 ) Write =====}
REQUEST_RESERVED = $F8; { Always set reserved bits =0 }
REQUEST_SET = $04; { Set DMA request }
REQUEST_CLR = $00; { Clear DMA request }
REQUEST_MSK = $03; { Specify channel in lower two bits }

{- Bits of channel masking register ( $0A, $D4 ) Write =====}
CHANNEL_RESERVED = $F8; { Always set reserved bits =0 }
CHANNEL_SET = $04; { Mask/lock DMA channel }
CHANNEL_CLR = $00; { Free DMA channel }
CHANNEL_MSK = $03; { Specify channel in lower two bits }

{- Bits of mode register ($0B, $D6) Write =====}
MODE_DEMAND = $00; { Transfer "On Call" }
MODE_SINGLE = $40; { Transfer single values }
MODE_BLOCK = $80; { Block transfer }
MODE_CASCADE = $C0; { Transfer cascaded }
MODE_DECREMENT = $20; { Decrement }
MODE_AUTOINIT = $10; { Auto initialization after end }
MODE_VERIFY = $00; { Verify }
MODE_WRITE = $04; { Write to memory }
MODE_READ = $08; { Read from memory }
MODE_INVALID = $0C; { Invalid }
MODE_CHANNELMSK = $03; { Specify channel in lower two bits }

Procedure dma_MasterClear( iChan : Integer );

Procedure dma_SetRequest( iChan : Integer );

Procedure dma_ClrRequest( iChan : Integer );

Procedure dma_SetMask( iChan : Integer );

Procedure dma_ClrMask( iChan : Integer );

Function dma_ReadStatus( iChan : Integer ) : Byte;

Procedure dma_ClrFlipFlop( iChan : Integer );

Function dma_ReadCount( iChan : Integer ) : Word;

Procedure dma_SetChannel( iChan : Integer;
lpMem : Pointer;
uSize : Word;

```

```

bMode : Byte );

Procedure dma_Mem2Mem(  bPage : Byte;
                        uSource, uDest,  uSize : Word );

Function dma_Until64kPage( lpMem : Pointer; uSize : Word ) : Word;

Procedure dma_Free( lpMem : Pointer );

Function dma_AllocMem( uSize : Word ) : Pointer;

{-----}

Implementation

uses DOS;
Const
dma_address      : array[0..7] of Byte=($00,$02,$04,$06,$C0,$C4,$C8,$CC);
dma_count        : array[0..7] of Byte=($01,$03,$05,$07,$C2,$C6,$CA,$CE);
dma_page         : array[0..7] of Byte=($87,$83,$81,$82,$88,$8B,$89,$8A);

{-- Register offsets for Master and Slave -----}
dma_status       : array[0..1] of Byte=($08,$D0); { Status register [Read] }
dma_command      : array[0..1] of Byte=($08,$D0); { Command register [Write] }
dma_request      : array[0..1] of Byte=($09,$D2); { Trigger DMA request }
dma_chmask       : array[0..1] of Byte=($0A,$D4); { Mask channels separately }
dma_mode         : array[0..1] of Byte=($0B,$D6); { Transfer mode }
dma_flipflop     : array[0..1] of Byte=($0C,$D8); { Address/Counter Flip-flop }
dma_masterclr    : array[0..1] of Byte=($0D,$DA); { Reset controller }
dma_temp         : array[0..1] of Byte=($0D,$DA); { Temporary register }
dma_maskclr      : array[0..1] of Byte=($0E,$DC); { Free all channels }
dma_mask         : array[0..1] of Byte=($0F,$DE); { Mask channels together }

{*****}
{ dma_Masterclear : Reset controller of a channel }
{*****}
{-----*}
{ Input : iChan : Number of channel (0-7) }
{*****}
Procedure dma_MasterClear( iChan : Integer );

Begin
    iChan := iChan and $0007;
    port[dma_masterclr[iChan div 4]] := 0 ;
End;

{*****}
{ dma_SetRequest : Set transfer to specified channel }
{*****}
{-----*}
{ Input : iChan : Number of channel (0-7) }
{*****}
Procedure dma_SetRequest( iChan : Integer );

Begin
    iChan := iChan and $0007;
    port[dma_request[iChan div 4]] := REQUEST_SET or ( iChan and $03 );
End;

{*****}
{ dma_ClrRequest : Stop transfer to specified channel }
{*****}
{-----*}
{ Input : iChan : Number of channel (0-7) }
{*****}
Procedure dma_ClrRequest( iChan : Integer );

Begin
    iChan := iChan and $0007;
    port[dma_request[iChan div 4]] := REQUEST_CLR or ( iChan and $03 );
End;

{*****}
{ dma_SetMask : Mask (lock) specified channel }
{*****}
{-----*}
{ Input : iChan : Number of channel (0-7) }
{*****}
Procedure dma_SetMask( iChan : Integer );

Begin
    iChan := iChan and $0007;
    port[dma_chmask[iChan div 4]] := CHANNEL_SET or ( iChan and $03 );
End;

{*****}
{ dma_ClrMask : Free specified channel }
{*****}
{-----*}
{ Input : iChan : Number of channel (0-7) }
{*****}

```

```

Procedure dma_ClrMask( iChan : Integer );

Begin
    iChan := iChan and $0007;
    port[dma_chmask[iChan div 4]] := CHANNEL_CLR or ( iChan and $03 );
End;

{*****}
{ dma_ReadStatus : Read status of controller belonging to          }
{                  specified channel                                }
{*****}
{-----*}
{ Input : iChan : Number of channel (0-7)                          }
{ Output : Status of controller                                    }
{*****}
Function dma_ReadStatus( iChan : Integer ) : Byte;

Begin
    iChan := iChan and $0007;
    dma_ReadStatus := port[dma_status[iChan div 4]];
End;

{*****}
{ dma_ClrFlipFlop : Clear flip-flop of controller of              }
{                  specified channel                                }
{-----*}
{ Input : iChan : Number of channel (0-7)                          }
{-----*}
{ Info : The Flip-flop is used to differentiate the LO-byte and HI-byte }
{        of the transfer address or the transfer counter.          }
{-----*}
Procedure dma_ClrFlipFlop( iChan : Integer );

Begin
    iChan := iChan and $0007;
    port[dma_flipflop[iChan div 4]] := 0;
End;

{*****}
{ dma_ReadCount : Read transfer counter of specified channel      }
{-----*}
{ Input : iChan : Number of channel (0-7)                          }
{ Output : Current transfer counter (0-65535)                      }
{-----*}
Function dma_ReadCount( iChan : Integer ) : Word;

var l, h : Byte;

Begin
    iChan := iChan and $0007;

    dma_ClrFlipFlop( iChan );
    l := port[dma_count[iChan]];
    h := port[dma_count[iChan]];
    dma_ReadCount := h * 256 + l;
End;

{*****}
{ dma_SetChannel : Prepare DMA channel for transfer               }
{-----*}
{ Input : iChan : Number of channel (0-7)                          }
{         lpMem : Address of memory                                }
{         uSize : Number of bytes to be transferred                }
{         bMode : Transfer mode                                    }
{         bAlign : DMA_BIT8 or DMA_BIT16 transfer                 }
{-----*}
{ Info : To enable 16 bit transfers, the linear address          }
{        passed to the DMA controller must be multiplied by 2.    }
{        With 16 bit transfers, up to 128K of data                }
{        can be transferred.                                       }
{-----*}
Procedure dma_SetChannel( iChan : Integer;
                        lpMem : Pointer;
                        uSize : Word;
                        bMode : Byte );

var uAddress : Word;
    bPage : Byte;

Begin
    iChan := iChan and $0007; { Max. 8 DMA channels }

    dma_SetMask( iChan ); { DMA transfers 1 more byte than specified! }
    { transfer at least 1 byte (0==1) }
    if uSize <> 0 then uSize := uSize - 1;

    {- generate linear 20 bit address -----}

```

```

if iChan <= 3 then
  Begin
    { Address = lower 16 bits of 20 bit address }
    uAddress := Word ( ( Longint ( lpMem ) and $FFFF0000 ) shr 12 )
                + ( Longint ( lpMem ) and $FFFF ) ;
    { Page = upper 4 bits of 20 bit address }
    bPage := Byte ( ( ( ( Longint ( lpMem ) ) and $FFFF0000 ) shr 12 )
                  + ( Longint ( lpMem ) and $FFFF ) ) shr 16 ) ;
  End
else
  { 16 bit DMA }
  Begin
    { Address = lower 16 bits of 20 bit address }
    uAddress := Word ( ( Longint ( lpMem ) and $FFFF0000 ) shr 13 )
                + ( ( Longint ( lpMem ) and $FFFF ) shr 1 ) ;
    { Page = upper 4 bits of 20 bit address }
    bPage := Byte ( ( ( ( Longint ( lpMem ) and $FFFF0000 ) shr 12 )
                  + ( Longint ( lpMem ) and $FFFF ) ) shr 16 ) ;
    bPage := bpage and $FE;
    uSize := usize div 2; { Counting words, not bytes! }
  End;

port[dma_mode[iChan div 4]] := bMode or ( iChan and MODE_CHANNELMSK );

dma_ClrFlipFlop( iChan ); { Clear address/counter flip-flop and... }
{ send address to DMA controller (LO-byte/HI-byte) }
port[dma_address[iChan]] := LO( uAddress );
port[dma_address[iChan]] := HI( uAddress );
port[dma_page[iChan]] := bPage; { Set memory page }

dma_ClrFlipFlop( iChan ); { Clear address/counter flip-flop and ... }
{ send counter to DMA controller (LO-byte/HI-byte) }
port[dma_count[iChan]] := LO( uSize );
port[dma_count[iChan]] := HI( uSize );

dma_ClrMask( iChan ); { Free DMA channel again }
End;

{*****}
{ dma_Mem2Mem : Shift memory block within a 64K page }
{*****}
{-----*}
{ Input : bPage : Address of 64K page }
{         uSource : Source }
{         uDest : Destination }
{         uSize : Number of bytes to be transferred }
{-----*}
{ Note: Memory to memory DMA transfer has not been }
{ implemented with newer computers (80286 and above). In Protected Mode }
{ or Virtual 8086 Mode of the processor, DMA0 }
{ accesses are generally intercepted and cause }
{ the system to crash. }
{*****}

Procedure dma_Mem2Mem( bPage : Byte;
                      uSource, uDest, uSize : Word );

Begin
  dma_MasterClear( 0 ); { Reset master controller via channel 0 }
                        { DMA transfers 1 more byte than specified! }
                        { transfer at least 1 byte (0==1) }
  if uSize<>0 then uSize :=uSize - 1 ;

  port[dma_address[0]] := LO( uSource );
  port[dma_address[0]] := HI( uSource );
  port[dma_page[0]] := bPage; { Set memory page }

  port[dma_address[1]] := LO( uDest );
  port[dma_address[1]] := HI( uDest );
  port[dma_page[1]] := bPage; { Set memory page }

  { Send counter to DMA controller (LO-byte/HI-byte) }
  port[dma_count[0]] := LO( uSize );
  port[dma_count[0]] := HI( uSize );

  { dma_ClrFlipFlop( 1 ); Clear address/counter flip-flop and ... }
  port[dma_count[1]] := LO( uSize );
  port[dma_count[1]] := HI( uSize );

  { Setting COMMAND_ADH0 doesn't increment the address register of channel 0. }
  { Thus, the memory block which channel 2 processes }
  { is initialized with a value. }
  port[dma_command[0]] := COMMAND_MEM2MEM or
                        COMMAND_FIXEDPRI or
                        COMMAND_COMPRESS;

  { When the memory blocks overlap, the program must differentiate between incrementation }
  { and decrementation !> (MODE_DECREMENT). With }
  { decrementing, the memory address + the number of bytes to be copied must be passed }
  { to the specific DMA channel as the address! }
  port[dma_mode[0]] := MODE_BLOCK or MODE_READ or 0; { Channel 0 }

```

```

port[dma_mode[0]] := MODE_BLOCK or MODE_WRITE or 1; { Channel 1 }

dma_SetRequest( 0 );
End;

{*****}
{ dma_Until64kPage : Determine number of bytes of a passed memory }
{                      area within its 64K page. }
{*****}
{-----*}
{ Input : lpMem : Address of memory to be checked }
{          uSize : Size of memory area }
{ Output : Number of bytes within memory until }
{          end of memory is reached or until next 64K page }
{*****}
Function dma_Until64kPage( lpMem : Pointer; uSize : Word ) : Word;

var lAdr, lPageEnd : Longint;

Begin
{-- Determine 20 bit linear address from far pointer -----}
lAdr := ( Longint ( Longint ( lpMem ) shr 16 ) shl 4 ) +
        ( Longint ( lpMem ) and $ffff );
lPageEnd := lAdr + $FFFF; { add 1 page }
{ Determine bytes to end of page }
lPageEnd := lPageEnd and $FFFF0000;
lPageEnd := lPageEnd - lAdr;

if lPageEnd > uSize then dma_Until64kPage := uSize
else dma_Until64kPage := lPageEnd;
End;

{*****}
{ dma_FreeMem : Free DMA capable memory. }
{-----*}
{ Input : uSize : Size of allocated memory. }
{*****}
Procedure dma_Free( lpMem : Pointer);

var regs : Registers;

Begin
{ Get segment and free via DOS-Call }
regs.ah := $49;
regs.es := Longint ( lpMem ) shr 16;
msdos( regs );
End;

{*****}
{ dma_AllocMem : Allocate DMA capable memory. }
{-----*}
{ Input : uSize : Size of memory to be allocated (max. 64k). }
{ Output : Address of allocated memory. }
{-----*}
{ Info : - This function continues allocating memory until }
{          the memory it finds is located within a physical 64K limit. }
{          Note: The allocation strategy is NOT }
{          optimum (see HeapWalk). }
{          The allocated memory begins at a segment address }
{          and is thus page or 16 byte aligned. }
{*****}
Function dma_AllocMem( uSize : Word ) : Pointer;

Const
uSegment      : Word = $FFFF;
uOldSegment   : Word = $FFFF;
var
test : Longint;
regs : Registers;
Begin
{ Can memory still be allocated ? }

Repeat
regs.ah := $48;
regs.bx := uSize div 16 + 1;
msdos( regs );

if ( regs.flags and 1 ) = 0 then
Begin
usegment := regs.ax;

if uOldSegment <> $FFFF then { free old memory }
dma_Free( Pointer ( uOldSegment shl 16 ) );
uOldSegment := uSegment; { Note new memory }
{ is new memory located in 64K page ? }

if dma_Until64kPage(
Pointer ( Longint ( usegment ) shl 16), uSize ) = uSize then
Begin

```

```
        DMA_AllocMem := Pointer (Longint ( uSegment ) shl 16 );
        Exit;
    End;
End;
Until regs.flags > 0;
if uSegment <> $FFFF then
    dma_Free( Pointer ( uOldSegment shl 16 ) );
    DMA_AllocMem := NIL;
End;

End.
```