

```

/*****
/*          D M A U T I L . C          */
/**-----**/
/* task          : Makes functions available for programming the */
/*          DMA controller.          */
/**-----**/
/* author          : Michael Tischer / Bruno Jennrich          */
/* developed on   : 3/20/1994          */
/* last update    : 4/07/1995          */
/**-----**/
/* COMPILER       : Borland C++ 3.1, Microsoft Visual C++ 1.5 */
/*****
#ifdef __DMAUTIL_C          /* DMAUTIL.C can also be #Included */
#define __DMAUTIL_C

/*-- Add include files -----*/
#include <dos.h>
#include <conio.h>
#include <stdlib.h>

#include "dmautil.h"

/*- global variables -----*/
/*- Arrays are used by channel to access the DMA controller */
INT dma_address [8] = {0x00,0x02,0x04,0x06,0xC0,0xC4,0xC8,0xCC};
INT dma_count   [8] = {0x01,0x03,0x05,0x07,0xC2,0xC6,0xCA,0xCE};
INT dma_page    [8] = {0x87,0x83,0x81,0x82,0x88,0x8B,0x89,0x8A};

INT dma_status  [2] = {0x08,0xD0};          /* status register [read] */
INT dma_command [2] = {0x08,0xD0};          /* command register [write] */
INT dma_request [2] = {0x09,0xD2};          /* release DMA request */

```

```

INT dma_chmask    [2] = {0x0A,0xD4};          /* mask channels individually
*/
INT dma_mode      [2] = {0x0B,0xD6};          /* transfer mode */
INT dma_flipflop  [2] = {0x0C,0xD8};          /* address-counter-flipflop
*/
INT dma_masterclr[2] = {0x0D,0xDA};          /* reset controller */
INT dma_temp      [2] = {0x0D,0xDA};          /* temporary register */
INT dma_maskclr   [2] = {0x0E,0xDC};          /* free all channels */
INT dma_mask      [2] = {0x0F,0xDE};          /* completely mask channels
*/

/*****
/
/*      dma_Masterclear      :      reset      controller      of      a      channel
*/
/**-----
**/
/*      input      :      iChan      :      channel      number      (0-7)
*/
/*****
/
VOID dma_MasterClear( INT iChan )
{ iChan &= 0x0007;
  outp( dma_masterclr[ iChan / 4 ], 0 );
}

/*****
/* dma_SetRequest : trigger transfer on the specified channel */
/**-----
/* input : iChan : channel number (0-7) */
/*****
VOID dma_SetRequest( INT iChan )

```

```

{ iChan &= 0x0007;
  outp( dma_request[ iChan / 4 ], REQUEST_SET | ( iChan & 0x03 ) );
}

/*****
/* dma_ClrRequest : stop transfer on the specified channel          */
/**-----*/
/* input : iChan : channel number (0-7)                            */
/*****/
.1 .4 cm 328 scan
VOID dma_ClrRequest( INT iChan )
{ iChan &= 0x0007;
  outp( dma_request[ iChan / 4 ], REQUEST_CLR | ( iChan & 0x03 ) );
}

/*****
/* dma_SetMask : mask (lock) specified channel                      */
/**-----*/
/* input : iChan : channel number (0-7)                            */
/*****/
VOID dma_SetMask( INT iChan )
{ iChan &= 0x0007;
  outp( dma_chmask[ iChan / 4 ], CHANNEL_SET | ( iChan & 0x03 ) );
}

/*****
/* dma_ClrMask : release specified channel                          */
/**-----*/
/* input : iChan : channel number (0-7)                            */
/*****/
VOID dma_ClrMask( INT iChan )
{ iChan &= 0x0007;

```

```

    outp( dma_chmask[ iChan / 4 ], CHANNEL_CLR | ( iChan & 0x03 ) );
}

/*****
/* dma_ReadStatus : read status of the controller for specified      */
/*                      channel                                          */
/**-----**/
/* input : iChan : channel number (0-7)                                */
/* output : controller status                                          */
/*****
BYTE dma_ReadStatus( INT iChan )
{
    iChan &= 0x0007;
    return ( BYTE )inp( dma_status[ iChan / 4 ] );
}

/*****
/* dma_ClrFlipFlop : Clear FlipFlop of controller for specified      */
/*                      channel                                          */
/**-----**/
/* input : iChan : channel number (0-7)                                */
/**-----**/
/* Info : The FlipFlop differentiates between LO and HI Byte        */
/*          in the transfer address, or transfer counter.            */
/*****
VOID dma_ClrFlipFlop( INT iChan )
{
    iChan &= 0x0007;
    outp( dma_flipflop[ iChan / 4 ], 0 );
}

/*****
/* dma_ReadCount : read transfer counter of specified channel        */
/**-----**/

```

```

/* input : iChan : channel number (0-7) */
/* output : current transfer counter (0-65535) */
/*****/
WORD dma_ReadCount( INT iChan )
{
    BYTE l, h;
    iChan &= 0x0007;

    dma_ClrFlipFlop( iChan );
    l = ( BYTE )inp( dma_count[ iChan ] );
    h = ( BYTE )inp( dma_count[ iChan ] );
    return MAKEWORD( h, l );
.l .4 cm 453 scan
}

/*****/
/* dma_SetChannel : prepare DMA channel for transfer */
/*****-----*/
/* input : iChan : channel number (0-7) */
/*          lpMem : memory address */
/*          uSize : number of bytes to be transferred */
/*          bMode : transfer mode */
/*          bAlign : DMA_BIT8 or DMA_BIT16 transfer */
/*****-----*/
/* Info : To make 16 bit transfers possible, the linear address */
/*          passed to the DMA controller must be multiplied by 2. */
/*          In 16 bit transfers, up to 128K of data can be transferred. */
/*          */
/*****/
VOID dma_SetChannel( INT iChan, LPVOID lpMem, WORD uSize,
                    BYTE bMode )
{
    WORD uAddress;
    BYTE bPage;

```

```

iChan &= 0x0007;                                /* Max. 8 DMA channels
*/

dma_SetMask( iChan );                            /*-block channel */
/* DMA transferred 1 byte more than specified! */
uSize = uSize ? uSize - 1 : 0;    /* transfer min. 1 byte (0==1) */

/*- create linear 20 bit address -----*/
if( iChan <= 3 )                                /* 8Bit DMA */
{
    /* address = lower 16 bit of the 20 bit address
*/
    uAddress = ( WORD ) ((( ( LONG ) lpMem ) & 0xFFFF0000L ) >> 12L ) +
                        ((( LONG ) lpMem ) & 0xFFFFL ));
    /* page = top 4 bit of the 20 bit address */
    bPage = ( BYTE ) ((( ( LONG ) lpMem ) & 0xFFFF0000L ) >> 12L ) +
                  ((( LONG ) lpMem ) & 0xFFFFL ) >> 16);
}
else                                             /* 16 bit DMA */
{
    /* address = lower 16 bit of the 20 bit address
*/
    uAddress = ( WORD ) ((( ( LONG ) lpMem ) & 0xFFFF0000L ) >> 13L ) +
                        ((( LONG ) lpMem ) & 0xFFFFL ) >> 1L ));
    /* page = top 4 Bit of the 20 bit address */
    bPage = ( BYTE ) ((( ( LONG ) lpMem ) & 0xFFFF0000L ) >> 12L ) +
                  ((( LONG ) lpMem ) & 0xFFFFL ) >> 16);
    bPage &= ( BYTE ) 0xFE;
    uSize /= 2;    /* WORDs, not BYTEs, are counted! */
}

outp( dma_mode[ iChan / 4 ], bMode | ( iChan & MODE_CHANNELMSK ) );

```

```

dma_ClrFlipFlop( iChan );    /* clear address/counter flipflop and...*/
    /* send address to the DMA controller (LO/HI-BYTE) */
outp( dma_address[ iChan ], LOBYTE( uAddress ) );
outp( dma_address[ iChan ], HIBYTE( uAddress ) );
outp( dma_page[ iChan ], bPage );    /* set memory page */

dma_ClrFlipFlop( iChan );    /* clear address/counter flipflop and ...*/
    /* send counter to the DMA controller (LO/HI-BYTE) */
outp( dma_count[ iChan ], LOBYTE( uSize ) );
outp( dma_count[ iChan ], HIBYTE( uSize ) );

dma_ClrMask( iChan );        /* clear DMA channel once again
*/
}

/*****
/* dma_Mem2Mem : shift memory block within a 64K page          */
**-----**
/* input : bPage      : address for the 64K page                */
/*          uSource    : source                                  */
/*          uDest      : goal                                    */
/*          uSize      : number of bytes to be transferred      */
**-----**
/* Note: Memory-memory DMA transfer is not implemented in the newer */
/*       computers (from 80286). In the processor's protected      */
/*       or virtual-8086-mode, DMA0                                */
/*       access is generally intercepted and will cause the        */
/*       system to crash.                                          */
**-----**
VOID dma_Mem2Mem( BYTE bPage, WORD uSource, WORD uDest, WORD uSize )
{
    dma_MasterClear( 0 );    /* reset master controller via channel 0 */

```

```

/* DMA transferred 1 byte more than specified! */
uSize = uSize ? uSize - 1 : 0; /* transfer min. 1 byte (0==1) */

outp( dma_address[ 0 ], LOBYTE( uSource ) );
outp( dma_address[ 0 ], HIBYTE( uSource ) );
outp( dma_page[ 0 ], bPage ); /* set memory page */

outp( dma_address[ 1 ], LOBYTE( uDest ) );
outp( dma_address[ 1 ], HIBYTE( uDest ) );
outp( dma_page[ 1 ], bPage ); /* set memory page */
.1 .4 cm 453 scan

/* send counter to the DMA controller (LO/HI-BYTE) */
outp( dma_count[ 0 ], LOBYTE( uSize ) );
outp( dma_count[ 0 ], HIBYTE( uSize ) );

/* dma_ClrFlipFlop( 1 ); Clear address/counter flipflop and ...*/
outp( dma_count[ 1 ], LOBYTE( uSize ) );
outp( dma_count[ 1 ], HIBYTE( uSize ) );
/* By setting COMMAND_ADH0, the address register on channel 0 */
/* is not incremented. The memory block that handles channel2 */
/* is, consequently, initialized with a value.
*/
outp( dma_command[ 0 ], COMMAND_MEM2MEM |
      COMMAND_FIXEDPRI |
      COMMAND_COMPRESS );

/* If the memory blocks overlap, you need to differentiate between
*/
/* incrementing and decrementing (MODE_DECREMENT). With */
/* decrementing, the memory address + the number of bytes */
/* to be copied must be passed to the respective DMA channel */

```



```

    /* as the address! */
    outp( dma_mode[ 0 ], MODE_BLOCK | MODE_READ | 0 );      /* channel 0
*/
    outp( dma_mode[ 0 ], MODE_BLOCK | MODE_WRITE | 1 );     /* channel 1
*/

    dma_SetRequest( 0 );
}

/*****
/* dma_Until64kPage : Determine number of bytes in a passed */
/*                      memory area within its 64k page.      */
/*-----*/
/* input : lpMem : Address for the memory that will be examined */
/*          uSize : size of the memory area                      */
/* output : Number of bytes within the memory up to             */
/*          memory end or the next 64-K-page                     */
*****/
WORD dma_Until64kPage( LPVOID lpMem, WORD uSize )
{
    ULONG lAdr, lPageEnd;

    /* identify 20Bit linear address from far counter
*/
    lAdr = ( ( LONG )FP_SEG( lpMem ) << 4L ) +
           ( LONG )FP_OFF( lpMem );

    lPageEnd = lAdr + 0xFFFFL;                                /* add 1 page */
    lPageEnd &= 0xFFFFF000L;                                  /* determine bytes to the end of the
page */
    lPageEnd -= lAdr;

    return min( ( WORD ) lPageEnd, uSize );
}

```

```

/*****
/* dma_AllocMem : allocate DMA capable memory. */
/**-----**/
/* input : uSize : size of the memory to be allocated (max. 64k). */
/* output : address for the allocated memory. */
/**-----**/
/* Info : - This function allocates memory, until the
/*          located memory lies within a physical boundary of 64K. */
/*          Note: The allocations strategy is NOT */
/*          optimal (s. HeapWalk). */
/*          The allocated memory starts at a segment address */
/*          and is therefore page aligned or 16 byte aligned. */
/*****
LPVOID dma_AllocMem( WORD uSize )
{ WORD uSegment = 0xFFFFU;
  WORD uOldSegment = 0xFFFFU;

  /* Can memory still be allocated ? */
  while( !_dos_allocmem( uSize / 16 + 1, &uSegment ) )
  {
    if( uOldSegment != 0xFFFFU ) /* release old memory */
      dma_Free( MK_FP( uOldSegment, 0 ) );
    uOldSegment = uSegment; /* acknowledge new memory
  /*
                                /* is the new memory located in 64K page?
  /*
    if( dma_Until64kPage( MK_FP( uSegment, 0 ), uSize ) == uSize )
      return MK_FP( uSegment, 0 );
  }
  if( uSegment != 0xFFFFU ) /* Unfortunately, no memory!
  /*
    dma_Free( MK_FP( uOldSegment, 0 ) );

```

```

    return NULL;
}

/*****
/* dma_FreeMem : release DMA capable memory.                                     */
/**-----**
/* input : uSize : size of allocated memory.                                     */
*****/
VOID dma_Free( LPVOID lpMem )
{
    /* Determine segment and release via DOS-Call */
    _dos_freemem( FP_SEG( lpMem ) );
}

#endif

```