NATIONAL INSTRUMENTS®
The Software is the Instrument®

Application Note 029

Programming DMA on PC/XT/AT Computers

T. Hayles and D. Potter

Introduction

Direct Memory Access (DMA) is a useful and powerful mechanism for transferring data in data acquisition and control applications. A personal computer equipped with a DMA controller can transfer large amounts of data at high speeds between an I/O device and system memory. Programming a DMA operation generally consists of two tasks: programming the I/O device to use DMA, and programming the DMA controller. Programming an I/O device for DMA transfers is relatively simple. For example, a data acquisition board is programmed to collect data as usual. Typically, the only additional programming required is to set a configuration bit which enables the generation of DMA requests when data is available to be transferred. On the other hand, programming a DMA controller is more involved and most available documentation for programming DMA controllers is difficult to decipher. This application note explains how DMA works in PC/XT/AT computers and discusses how to program the 8237A DMA controller that is used in these systems. An example program is included to demonstrate the specifics of DMA programming.

Transferring Data with DMA

DMA is a method of transferring data at high speeds between an I/O device and memory. A specialized processor, the DMA controller, is used to transfer the data, bypassing the CPU. The DMA controller requests control of the system bus from the CPU, transfers the data, and relinquishes the bus to the CPU at the completion of the data transfer.

Besides DMA, there are two other methods used to transfer data in data acquisition applications: *processor interrupts* and *software polling routines*. With processor interrupts, the CPU is interrupted from the main execution program whenever data is available to be transferred. The CPU then branches to a special routine that transfers the data. With software polling, the CPU continually checks or polls the status of the I/O device for availability of data. DMA is inherently faster than both interrupts and polling because the CPU is not needed to fetch and execute data transfer instructions. The DMA controller is a peripheral device dedicated to the task of transferring data. Only one or two bus read/write cycles are required per transfer, and the DMA controller can respond very quickly to requests for attention from the I/O device. Also, the DMA controller offloads the burden of data transfer from the processor. DMA can proceed as a background task of the computer system, freeing the CPU to operate on other foreground tasks. In some applications, however, it may be desirable to involve the CPU directly in the transfer activity. If incoming data must be examined and processed on a point-by-point basis in real time, interrupts or polling should be used. DMA is better suited for transferring blocks of data.

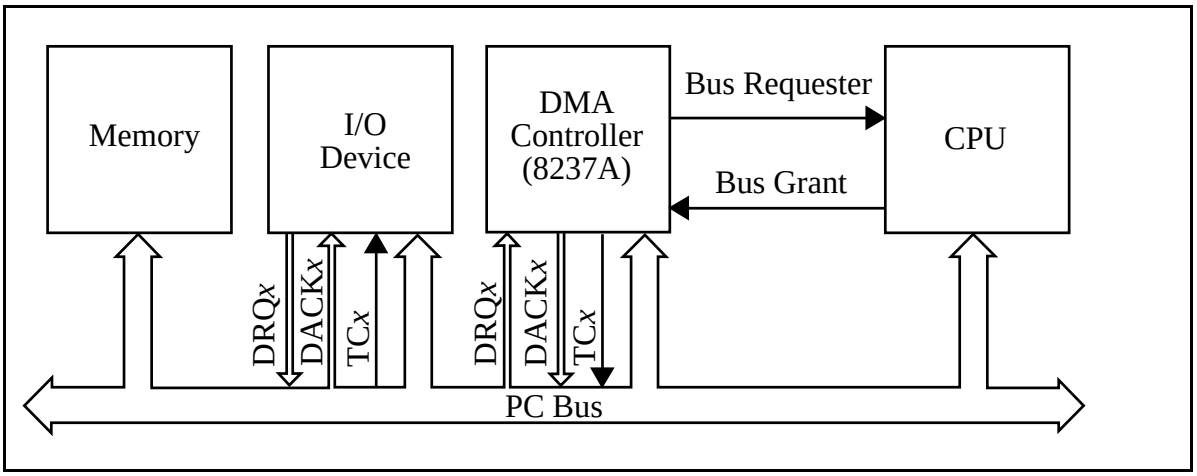

Figure 1. System Implementation of DMA

Figure 1 shows a generalized model of a PC/XT/AT system that uses DMA to transfer data between an I/O device and memory. The I/O device communicates with the DMA Controller using DMA request and acknowledge signals (DRQ*x* and DACK*x*, where *x* is the channel number). The DMA controller in turn communicates with the processor via bus request and acknowledge signals.

Figure 2 illustrates the general protocol of a DMA transfer of data from an I/O device to memory. First, the DMA controller is programmed with the starting address of the block of memory to be transferred and the number of transfers to be performed. The I/O device is configured to assert DMA requests whenever data is available to be transferred to memory. The I/O device requests a transfer by asserting a hardware DMA request signal (DRQ*x*). The DMA controller responds by asserting a hardware bus request signal to the CPU. Once the CPU grants control of the bus to the DMA controller (through a hardware acknowledge signal), the DMA controller places the programmed memory address on the address bus and asserts the DMA acknowledge signal (DACK*x*). This signals the I/O device to place the data onto the bus and remove the DMA request signal. The data is then latched into memory by the DMA controller. The DMA controller decrements its internal count register and updates its internal address register. This sequence of events is repeated until the transfer count reaches zero, referred to as *terminal count* (TC). At this point, the DMA controller asserts the hardware TC signal that signals the CPU and I/O device that the DMA transfer is finished.

In this example, the I/O device removes the DMA request signal and the DMA controller relinquishes the bus at the completion of each individual data transfer. This is referred to as *single* transfer mode. Alternatively, the DMA controller can perform multiple DMA transfers once it has gained control of the bus. In *demand* transfer mode, the DMA controller performs multiple DMA transfers as long as the I/O device continues to assert the DMA request. In *block* transfer mode, the DMA controller performs the entire preprogrammed DMA sequence in response to a single DMA request. While both demand and block transfer modes of DMA are faster than single-mode DMA, they must be used with caution. On PC/XT/AT computers, the I/O bus must be available every 15 µs for memory-refresh cycles. This is why block-mode DMA is not normally used on the PC/XT/AT and why demand-mode DMA devices must be designed to relinquish the bus at least once every 15 µs.

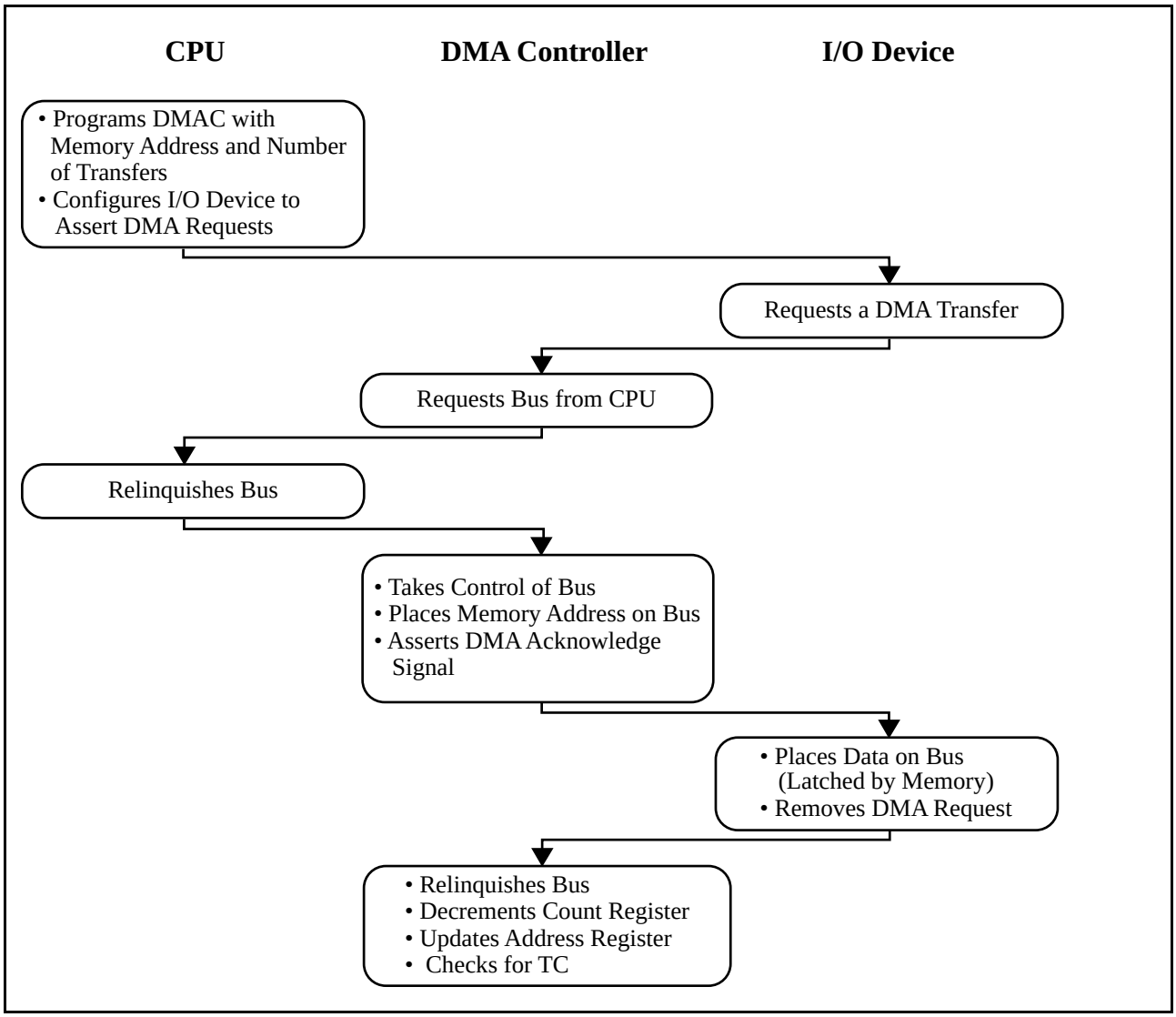

Figure 2. Sequence of Events in DMA Transfer

The type of DMA operation diagrammed in Figure 2 is referred to as *flyby* DMA because the DMA controller provides the preprogrammed memory address and the I/O device reads or writes the data directly to memory in a single bus cycle. Flyby DMA is faster than the second type of DMA, *fetch-and-deposit*. Fetch-and-deposit DMA transfers consist of two bus cycles: a read from the I/O device and a write to memory. For a more complete discussion of the different types and modes of DMA used on different computer platforms, see Application Note 011, *DMA Fundamentals on Various PC Platforms*.

DMA on PC/XT/AT Computers

The IBM PC design uses the Intel 8237A DMA controller, which provides four channels of DMA. The 8237A has 16-bit memory-address and byte-count registers for each DMA channel. This DMA controller performs flyby DMA in either single, demand, or block mode.

The original PC design uses a single 8237A to provide four channels of 8-bit DMA (the PC/XT has an 8-bit bus). Three of the four available DMA channels are reserved for the floppy disk, the hard disk drive, and memory-refresh

cycles. The PC design adds an 8-bit page register to extend the 16-bit address to the 24-bit PC memory address. This page register does *not* automatically increment when the address register reaches full value (0xFFFF), referred to as a *page boundary*. This means that DMA transfer sequences cannot cross the 64-kilobyte page boundary without reprogramming the DMA controller.

The design of the IBM PC AT adds a second 8237A DMA controller that increases the total number of DMA channels to seven (the first channel of the second DMA controller is used to cascade the two DMA controllers). The first DMA controller provides DMA Channels 0 through 3. DMA channels are no longer used for the hard drive and memory refresh, leaving three channels of 8-bit DMA available. The second DMA controller adds three channels (Channels 5 through 7) of 16-bit DMA. Again, a page register holds the upper byte of the memory address, but because Channels 5 through 7 are 16-bit channels, a single DMA sequence is limited to 64-kiloword transfers.

The clock frequency for the 8237A is usually derived from the system clock, varying from 3 MHz to 5 MHz in different systems. A single-mode DMA transfer needs five bus cycles for the data transfer plus a few more cycles for the transfer of bus control. The exact number of cycles needed to transfer bus control will depend on the system and the amount of bus traffic. Practical maximum throughput rates for single-mode DMA transfers will typically be on the order of 300,000 to 500,000 transfers per second.

Programming the 8237A DMA Controller

The 8237A DMA controller has three registers for each DMA channel and seven general-purpose registers. The I/O addresses for both 8237A DMA controllers in a PC/XT/AT computer are listed in Table 1.

Table 1. 8237A DMA Controller I/O Address Map

Register	**Address per Channel**						
	0	**1**	**2**	**3**	**5**	**6**	**7**
Base Address Register	0000	0002	0004	0006	00C4	00C8	00CC
Transfer Count Register	0001	0003	0005	0007	00C6	00CA	00CE
Page Register	0087	0083	0081	0082	008B	0089	008A
Command/Status Register	0008				00D0		
Mode Register	000B				00D6		
Clear Byte Pointer Register	000C				00D8		
DMAC Reset Register	000D				00DA		
Clear Mask Register	000E				00DC		
Single Mask Register	000A				00D4		
Write All Mask Register	000F				00DE		

Each DMA channel has a 16-bit Base Address Register and an 8-bit Page Register, which together are programmed with the starting address of the memory block to be written to or read from. When programming DMA Channels 0 through 3 (for 8-bit transfers), the Base Address Register is programmed with the lower 16 bits of the 20-bit *physical* address, while the Page Register is programmed with the upper four bits. If programming DMA Channels 5 through 7 (for 16-bit transfers), the physical address is first shifted one bit to the right (equivalent to dividing by two), and the least significant bit of the Page Register is always set to zero. Figure 3 illustrates the programming of the Base Address and Page Registers for both 8-bit and 16-bit transfers.

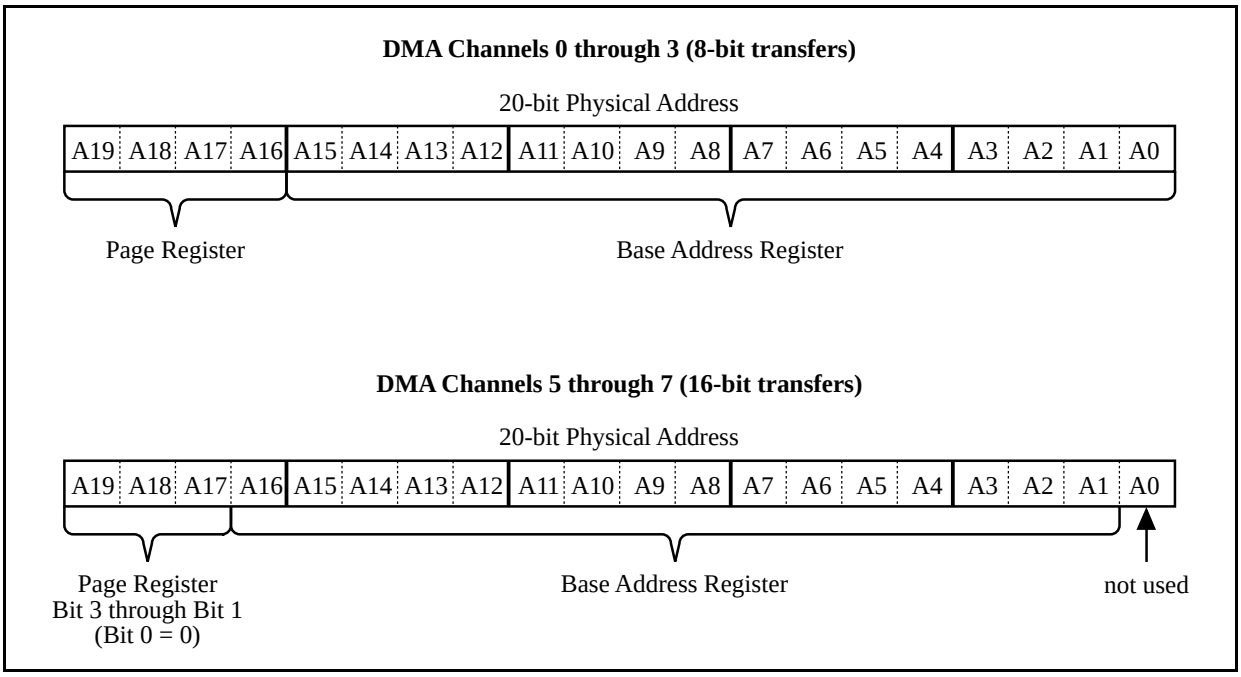

Figure 3. Programming Base Address and Page Registers

Again, it is important to remember that the Page Register does not automatically increment, and must be reprogrammed at each page boundary. Therefore, the maximum number of DMA transfers that can be performed without reprogramming the DMA controller is equal to the size of the Base Address Register, $2^{16} = 65{,}536$. The actual number is usually less because the lower 16 bits of the physical starting address are rarely all zero. There are two methods of working around this limitation. First, the DMA controller can be programmed so that TC is reached at the page boundary. When TC occurs, an interrupt routine programs the DMA controller to resume data transfer after the page boundary. The drawback of this method is that no DMA transfers are performed while the DMA channel is reprogrammed. This may cause loss of data in a data acquisition application. The second method, referred to as *dual DMA*, uses two DMA channels and special circuitry on the I/O board to perform DMA across page boundaries with no pause in DMA services. The first DMA channel is programmed to transfer data from the beginning of the buffer to the location of the page boundary. Before the transfer starts, a second DMA channel is also programmed to begin transferring at the location of the DMA page boundary. The I/O board must implement special DMA circuitry to support dual-DMA mode. DMA requests are generated by the I/O board on the first DMA channel until the I/O board senses the TC signal asserted by the DMA controller at the first page boundary. At this point, the I/O board stops asserting DMA requests on the first DMA channel and begins asserting DMA requests on the second programmed DMA channel. Therefore, there is no pause in DMA transfers while the controller is being reprogrammed. If the DMA data buffer contains more than one page boundary, then the DMA channels are alternated, each one being reprogrammed as the other channel is used to transfer data. National Instruments has a line of high-speed multifunction and digital I/O boards which implement this dual-DMA technique.

Each DMA channel also has a 16-bit Transfer Count Register, which is programmed with the total number of DMA transfers to be done, minus one. When the Transfer Count Register is read, it returns the current transfer count, which is decremented after each DMA transfer. Note that even if the Page Register were able to auto-increment, the Transfer Count Register would still impose a 65,536-transfer limit.

Both the Transfer Count Register and the Base Address Register must be programmed eight bits at a time using the Clear Byte Pointer Register. When the Clear Byte Pointer Register is written to, an internal pointer used to point to the upper or lower byte of the 16-bit Address and Count Register is cleared. The following 8-bit write to the Base Address Register or Transfer Count Register will address the lower byte of the register. The write operation automatically increments the internal pointer, and the next 8-bit write is addressed to the upper byte of the register.

Each DMA group also has a write-only Command Register and a read-only Status Register. However, these two registers are rarely if ever needed when programming the 8237A on PC/XT/AT computers. The Command Register sets the signal polarity of the DRQ*x* and DAK*x* lines, sets the group arbitration priority scheme to fixed or rotating, and can be used to enable or disable the channel group. The Status Register identifies channels that have pending DMA requests and channels that have reached TC. Bits 0 through 3 are set every time the respective channel reaches TC. These bits are cleared on each read of the Status Register. Bits 4 through 7 are set whenever the respective channel requests DMA service.

The Single Mask Register and Write All Mask Register are used to enable and disable DMA channels. The Single Mask Register is used to enable and disable individual DMA channels, while the Write All Mask Register can be used to mask the entire channel group. The Clear Mask Register, when written to, clears the Command and Status Registers, disables DMA requests, and executes a Clear Byte Pointer command.

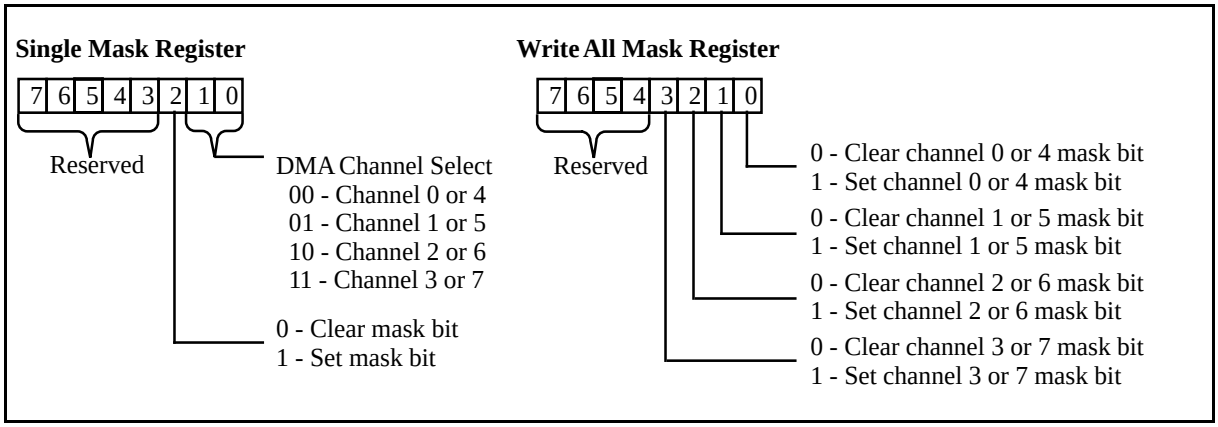

Figure 4. Single Mask and Write All Mask Register Descriptions

Each channel group has a Mode Register, which sets the DMA mode and direction, enables and disables auto-initialization, and determines whether the address register is incremented or decremented. Bits 2 and 3 set the type of transfer as a read (from memory to I/O device) or as a write (from I/O device to memory). Bit 4 of the Mode Register is to used to enable auto-initialization. When auto-initialization is enabled, the DMA controller is automatically reinitialized with the starting base address and transfer count when TC is reached. Auto-initialization is useful for implementing circular memory-buffer schemes used for double-buffered data acquisition. Bits 6 and 7 of this register set the DMA mode to single, demand, block, or cascade. Cascade mode is used only by DMA Channel 4 (for cascading Channels 0 through 3).

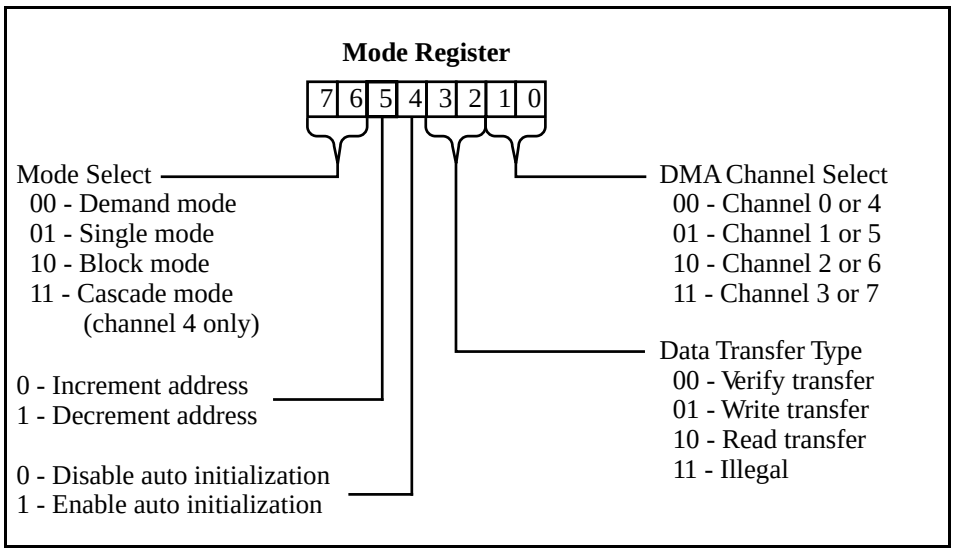

Figure 5. Mode Register Description

DMA Programming Example: DMAESP.C

This section contains a listing of the file DMAESP.C. This program is Microsoft C code that programs the PC/XT/AT DMA controller (the 8237A) to transfer a preprogrammed number of data points from an I/O device to memory. Also included in the listing is an example of a C program that calls the DMA routine.

The program checks the DMA buffer for the presence of a DMA page boundary. If a page boundary is detected, the routine simply returns an error code and exits without performing the DMA transfer. The user can modify the code to either relocate the buffer so that it does not contain a page boundary or perform DMA across the page boundary using one of the techniques discussed briefly in this application note.

```
/********************************* DMAESP.C *********************************
*  Engineering Software Package XT and AT DMA Controller Programming
*  Rev A.1
*  Copyright 1990 National Instruments Corporation.
*  All rights reserved.
*
*  This file contains code that will program a PC/XT/AT DMA controller.
*  It was written for the Microsoft C compiler.
*
****************************************************************************/

#include <dos.h>     /* For the macros FP_SEG() and FP_OFF() */

/*
 * The pageRegs array contains the I/O addresses of the DMA Page Registers.
 * The array is indexed using the DMA channel number (for example, pageRegs[2]
 * is the address of the Page Register for DMA Channel 2). The Page Registers
 * hold the most significant four bits of the 20-bit physical address used to
 * program the DMA controller. These Page Registers are static devices; that
 * is, the DMA controller cannot alter the contents of the Page Registers
 * during transfers. This means that the DMA controller cannot transfer data
 * across what is commonly called a "DMA page boundary". For 8-bit (byte)
 * transfers, a DMA page is 65,536 bytes in size. To transfer a full 8-bit DMA
 * page, the lower 16 bits of the 20-bit physical starting address must be
 * zero. For 16-bit (word) transfers, a DMA page is 131,072 bytes in size. To
 * transfer a full 16-bit DMA page, the lower 17 bits of the 20-bit physical
 * starting address must be zero.
 */
unsigned int pageRegs[] = {0x0087, 0x0083, 0x0081, 0x0082, 0, 0x008B, 0x0089,
                           0x008A};

/*
 * The countRegs and baseAddrRegs arrays contain the I/O addresses of the
 * Transfer Count Registers and the Base Address Registers, respectively.
 * These arrays are indexed using the DMA channel number similar to the
 * pageRegs[] array. The Transfer Count Register is programmed with the number
 * of bytes or words to transfer minus one. The Base Address Register is
 * programmed with the lower 16 bits of the 20-bit physical address. If word
 * transfers are to be done, the physical address is shifted one bit to the
 * right before programming the Base Address Register.
 */
unsigned int countRegs[] = {0x0001, 0x0003, 0x0005, 0x0007, 0, 0x00C6, 0x00CA,
                            0x00CE};
```

```
unsigned int baseAddrRegs[] = {0x0000, 0x0002, 0x0004, 0x0006, 0, 0x00C4,
                               0x00C8, 0x00CC};

/*
 * The following macros return the 16-bit I/O address of the desired register
 * for the desired DMA channel (ch refers to the DMA channel).
 */
#define   COUNT_reg(ch)          (countRegs[ch])
#define   BASE_ADDR_reg(ch)      (baseAddrRegs[ch])
#define   PAGE_ADDR_reg(ch)      (pageRegs[ch])
#define   CLR_BYTE_PTR_reg(ch)   ((ch < 4) ? 0x000C : 0x00D8)
#define   MODE_reg(ch)           ((ch < 4) ? 0x000B : 0x00D6)
#define   SNGL_MASK_reg(ch)      ((ch < 4) ? 0x000A : 0x00D4)

/**** The Mode Register bit map ****/
#define   SINGLE    (1<<6)
#define   BLOCK     (2<<6)
#define   CASCADE   (3<<6)

#define   ADDR_INC  0
#define   ADDR_DEC  (1<<5)

#define   AUTO_INIT_ENABLE   (1<<4)
#define   AUTO_INIT_DISABLE   0

#define   WRITE     (1<<2)  /* Input from an I/O port and WRITE to memory  */
#define   READ      (2<<2)  /* READ from memory and output to an I/O port  */

/*
 * This macro produces the DMA channel select bit mask that is ORed onto the
 * least significant two bits of commands written to the Mode Register.
 */
#define   SELECT_CHAN(ch)   ((ch < 4) ? ch : (ch-4))

/*
 * The following two macros are to be used with writes to the SNGL_MASK_reg.
 */
#define   ENABLE(ch)    ((ch < 4) ? ch : (ch - 4))
#define   DISABLE(ch)   ((1<<2)|((ch < 4) ? ch : (ch - 4)))

/*
 * The following are defined simply for the convenience of this example.
 * Remember that input receives data from an I/O device and WRITES it to
 * system memory. Likewise, output READS data from system memory and sends it
 * to an I/O device.
 */
#define INPUT  0
#define OUTPUT 1

/*
 * The error codes that DMARoutine() can return are defined.
 */
#define memErr  -1
#define noErr    0
```

```
/*************************************************************************
*   DMARoutine() programs the DMA controller on a PC/XT/AT to perform the type
*   of transfer indicated by the state of the calling parameters. Please
*   note that DMARoutine() does NOT enable the DMA channel after programming
*   it. Enabling the DMA channel should be done by the calling routine when
*   the caller is sure that the device that will generate the DMA requests is
*   driving the DMA request line. If the DMA channel is enabled while the
*   request line is in an unknown state, the controller may begin transferring
*   data prematurely.  The routine also checks if the transfer crosses a
*   physical DMA page boundary.  If one is found, an error is returned.
*************************************************************************/

int DMARoutine (dmaChan, startAddress, transferCount, autoInitMode, dir)
int
   dmaChan;           /* Channels 0, 1, 2, 3, 5, 6 or 7 */
void
   far *startAddress; /* A far pointer to an area in system memory */
unsigned int
   transferCount,     /* The number of bytes or words to transfer  */
   autoInitMode,      /* 0:auto initialization disabled,           */
                         1:auto init enabled                       */
   dir;               /* direction of DMA transfer;                */
                         0:input, 1:output                         */

{
    unsigned char
      lsb,            /* The least significant 8 bits              */
                         of the 20-bit address                     */
      msb,            /* The next significant 8 bits               */
                         of the 20-bit address                     */
      page,           /* The high 4 bits of the 20-bit             */
                         physical address                          */
      lcnt,           /* The least significant 8 bits              */
                         of the transfer count                     */
      hcnt,           /* The most significant 8 bits               */
                         of the transfer count                     */
      dmaMode;        /* Bit map describing the type               */
                         of DMA transfer                           */
   unsigned long
      addr;           /* The 20-bit physical address               */
                         of the DMA buffer                         */
   unsigned int
      maxTransferCnt; /* The number of transfers before page       */
                         boundary is encountered in DMA buffer     */

/*
 * Calculate the 20-bit physical address from the SEGMENT:OFFSET far pointer
 * by shifting the segment four bits left and adding the result to the offset.
 */
   addr = ((unsigned long)FP_SEG (startAddress) << 4) + FP_OFF (startAddress);

/*
 * Check whether DMA transfer crosses a physical DMA page boundary. The
 * maximum 8-bit DMA transfer count is 65,536 bytes and the maximum 16-bit DMA
 * transfer count is 65,536 16-bit words. The number of transfers from the
 * beginning of buffer to the page boundary is computed (maxTransferCnt) and
 * compared to the requested transfer count.
 */
```

```
   maxTransferCnt = 0x10000 - ((dmaChannel < 4 ? addr : addr>>1) & 0xffff);
   if (transferCount > maxTransferCnt)
      return memErr;
/*
 * DMA Channels 0 through 3 perform 8-bit transfers only. DMA Channels 5
 * through 7 perform 16-bit transfers only, and the least significant bit of
 * the address is not used.
 */
   if (dmaChan < 4)
   {
      lsb  = addr;
      msb  = addr >> 8;
      page = addr >> 16;
   }
   else /* dmaChan is 5, 6 or 7 */
   {
      lsb  = addr >> 1;
      msb  = addr >> 9;
      page = addr >> 16 & 0xfe;
   }

/*
 * The DMA controller transfers data until the contents of its Transfer Count
 * Register "roll over" from 0x0000 to 0xFFFF (it counts down, not up). So to
 * account for this "extra" transfer, subtract 1 from the desired transfer
 * count before programming this register.
 */
   --transferCount;
   lcnt = transferCount;
   hcnt = transferCount >> 8;

/*
 * The Mode Register bit map determines the type of request (SINGLE), the
 * direction, the DMA channel, whether the Address Register increments or
 * decrements (INC), and whether to use autoinitialize mode.
 *
 * If autoinitialize mode is enabled, the DMA controller reloads the initial
 * address (the startAddress) and transfer count when the contents of the
 * Transfer Count Register roll over. In this manner, data can be continuously
 * read from or written to the same area of memory.
 */
   dmaMode = SINGLE | ADDR_INC |
             (autoInitMode ? AUTO_INIT_ENABLE : AUTO_INIT_DISABLE) |
             (dir == INPUT ? WRITE : READ) | SELECT_CHAN(dmaChan);

/*
 * Now program the DMA controller. Always disable interrupts while writing to
 * the DMA controller's registers.
 *
 * The writes to the CLR_BYTE_PTR_reg are necessary to set the internal byte
 * pointer of the controller to the proper location for programming those
 * registers that are accessed using a single I/O address. The order in which
 * these registers are programmed is crucial to proper DMA operation.
 */
```

```
   _disable();
   outp (SNGL_MASK_reg(dmaChan)    , DISABLE(dmaChan));
   outp (MODE_reg(dmaChan)         , dmaMode);
   outp (CLR_BYTE_PTR_reg(dmaChan) , 0);
   outp (BASE_ADDR_reg(dmaChan)    , lsb);
   outp (BASE_ADDR_reg(dmaChan)    , msb);
   outp (PAGE_ADDR_reg(dmaChan)    , page);
   outp (CLR_BYTE_PTR_reg(dmaChan) , 0);
   outp (COUNT_reg(dmaChan)        , lcnt);
   outp (COUNT_reg(dmaChan)        , hcnt);
   _enable();
   return noErr;
}

/*************************************************************************
*  The following is an example of a C program that calls DMARoutine().
*************************************************************************/

/* Declare an area in memory to be used for the transfer. */
int far DMAbuffer[5000];

main()
{
   int err;
/*
 * Program the DMA controller to perform the desired transfer. These
 * parameters are for illustration purposes only. If DMAbuffer contains
 * a page boundary, then the routine returns an error.
 */
   err = DMARoutine (5, DMAbuffer, 5000, 0, INPUT);
   if (err != noErr)
      exit (-1);

/*
 * Code should be placed here that programs the I/O device, enabling the
 * DMA request mode of the device.
 *
 * Enable the DMA channel.
 */
   outp (SNGL_MASK_reg(5), ENABLE(5));

/*
 * Code should be placed here that triggers the I/O device to begin generating
 * DMA requests. When the I/O device has finished, be sure to disable the DMA
 * channel.
 */
    outp (SNGL_MASK_reg(5), DISABLE(5));
}
```

Conclusion

DMA is very effective for moving large amounts of data at high rates between I/O devices and memory with minimal CPU intervention. For this reason, DMA is very useful for data acquisition and control applications.

The PC/XT/AT computers use the Intel 8237A DMA controller chip. The original PC used a single 8237A to provide four 8-bit DMA channels. The PC/AT computer design added a second 8237A to provide three additional 16-bit DMA channels. The 8237A DMA controller implements flyby DMA in single, demand, or block mode. However, due to memory refresh requirements, demand and block mode are rarely used on PC/XT/AT computers.

For developers of driver-level software, the example program DMAESP.C (included in this application note) demonstrates how to program the 8237A DMA controller to transfer a block of data from an I/O device to memory. The code can be used for a data acquisition application by simply adding the code to configure the I/O device to collect data and generate DMA requests.

340230-01 Jan94