Using the SmartDMA™ Controller with USB on the Am186™CC Microcontroller

AMD

Application Note

by Mark Langsdorf

This application note explains how using the SmartDMA™ controller with USB enhances the functionality of an Am186™CC microcontroller. This application note assumes that the reader is knowledgeable about the Am186CC microcontroller and USB.

INTRODUCTION

The SmartDMA controller provides a powerful and efficient way to transfer data between memory and the USB data FIFOs on the Am186CC microcontroller. Correctly implemented, the SmartDMA controller enables full utilization of the 12-Mbit/s full-speed bus bandwidth of the 1.0, 1.1, and 2.0 USB specifications. The most efficient implementation of the SmartDMA controller is the buffer-per-IRP (I/O request packet), which requires complex software handling, either at the end of each buffer or for error recovery.

This application note discusses the differences between buffer-per-packet and buffer-per-IRP SmartDMA channels, and explains why a system can use either one. Also included are an example of how to set up a buffer-per-IRP BULK interface and an example of error recovery routines for the buffer-per-IRP BULK interface. The procedures and code in this application note must be combined with EPD CodeKits CK0005xx or CK0012xx to implement a fully functioning USB system using the SmartDMA controller. To access the CodeKits, refer to the following link on the AMD web site: www.amd.com/products/lpd/codekits/downloads.html. For more information about the SmartDMA controller and USB, refer to the *Am186™CC/CH/CU Microcontrollers User's Manual*, order #21914 and the *Am186™CC/CH/CU Microcontrollers Register Set Manual*, order #21916.

SmartDMA™ CONTROLLER MODES

When used with a USB, the SmartDMA controller on the Am186CC microcontroller can operate in two modes, buffer-per-packet (also known as store status) and buffer-per-IRP (also known as no-store status). These two modes describe the memory buffer that is pointed to by the descriptor in the SmartDMA channel descriptor ring.

Buffer-Per-Packet

In buffer-per-packet mode, each USB packet has its own descriptor. When a packet is transferred, the descriptor is returned to software control by the hardware, and the SmartDMA channel pair Current Buffer Descriptor (SDxCBD) register is automatically updated to point to the next available buffer. If an error occurs, a NAK is issued to terminate the transaction, and software must intervene (in some cases) to reset the SmartDMA channel. Buffer lengths are limited to the size of USB packets.

Buffer-Per-IRP

In buffer-per-IRP mode, several USB data packets can share the same descriptor. The advantage of buffer-per-IRP is that there is no limit to the size of each buffer. So, several hundred bytes or more can be transferred between each interrupt. With buffer sizes of approximately one kilobyte, the Am186CC microcontroller can send data at the full bus rate of 12 Mbit/s.

The buffer-per-IRP method of data transfer is faster and more efficient than buffer-per-packet because buffer-per-IRP minimizes the overhead. Both methods generate an interrupt after each buffer, and the interrupts for each method take the same amount of time to complete. The SmartDMA controller cannot transfer data while the processor is handling an interrupt, so data transfer is faster if fewer interrupts occur. Because the buffer-per-IRP method generates only one interrupt per IRP and an IRP can contain several packets, only one interrupt occurs per several packets, which is considerably less than with the buffer-per-packet method that generates one interrupt per packet. Therefore, the buffer-per-IRP method of data transfer is faster and more efficient.

SETTING UP THE USB AND SmartDMA™ CONTROLLER

The following steps describe how to set up the USB endpoints and the SmartDMA controller on the Am186CC microcontroller.

Publication# **23112** Rev: **A** Amendment/**0**
Issue Date: **November 1999**

Setting Up the USB Endpoint

1. Only the USB data endpoints can be used with the SmartDMA controller. Endpoint A must be configured for OUT data on SmartDMA Channel 2 (receive), while Endpoint B must be configured for IN data on the SmartDMA Channel 2 (transmit). Endpoint C must be configured for OUT on Channel 3 (receive), and Endpoint D must be IN on Channel 3 (transmit). Depending on the application, the appropriate endpoint should be chosen.
2. The endpoint should be disabled and then defined.
   a. Definition register 1 (xEPDEF1) controls the endpoint number, the USB configuration, interface, setting (as set by the host during configuration), direction (IN or OUT), and the type, which can be bulk, interrupt, or isochronous.
   b. Definition register 2 (xEPDEF2) controls the FIFO size and the maximum packet size. The FIFO size is only relevant for IN buffers. Unless the NOT_ZERO or NOT_LAST_BYTE bits of the endpoint control register are cleared, the USB controller responds to host requests with NAK responses if the FIFO is not completely full.
   c. Definition register 3 (xEPDEF3) controls the SmartDMA channel mode and operation. For a buffer-per-IRP transfer, set the transfer mode to 100b, clear all of the IMSK bits, and set the BUFF_ERR_SMSK and OTH_ERR_SMSK bits.

      These actions cause the endpoint to stop on any error, generating an interrupt in the process. If the endpoint operates in the OUT direction, the SHRT_PKT_SMSK bit should also be set. This action causes the endpoint to stop at the end of the IRP, generating an interrupt. For buffer-per-packet transfer, set the mode to 101b and set the IMSK bits. These actions cause the endpoint to interrupt when an error occurs or a complete packet is received or sent.
3. Set interrupt vector 0xC to point to the USB interrupt handler.
4. Set interrupt channel 2 for an internal, level-triggered source, and then enable it with the CH2CON register.

Setting Up the SmartDMA™ Channel

1. Create and initialize a buffer of descriptors in memory.
   a. Allocate an array of data buffer descriptors and a data buffer in memory.
   b. Configure the buffer descriptors by setting the LADR and HADR bytes to point to the data buffer, and then set the OWN bit for any buffers that are available for transfer.
   c. Set the STP and ENP bits at the start and end (respectively) of any buffer chains. If each SmartDMA buffer contains all the data for a transfer, set both the STP and ENP bits in each buffer.
2. Program the SmartDMA control register (SDxCON).
   a. Clear the start bit (xXST), and set the DSEL bit. This stops the channel and makes USB the requesting source.
   b. Clear the xXSO bit so that the hardware always returns the descriptor to software control when the hardware transfers BCNT bytes.
   c. For IN/transmit endpoints (B or D), set the TTCI bit to cause an interrupt after all the data is transferred.
   d. For OUT/receive endpoints (A or C), do not set any other bits.
3. Write the descriptor ring address into the appropriate Ring Address High (SDxxRAH) and Ring Address Low (SDxxRCAL) registers.
4. Set the Current Buffer Descriptor (SDxCBD) register to point to the first available buffer.
5. Turn on the SmartDMA channel.
6. For receive endpoints, turn on the USB ACT_REQ interrupt mask.
7. For transmit endpoints, do not turn on the USB interrupt mask or the SmartDMA channel interrupt mask until data is available.

   Turning on the USB interrupt mask or the SmartDMA channel interrupt mask before data is available can cause unnecessary buffer error interrupts to occur when the FIFO underruns.

ERROR HANDLING

Error Handling in Buffer-Per-Packet Mode

In buffer-per-packet mode, error handling is relatively easy for IN endpoints and OTHER_ERRs on OUT endpoints. Each packet is in a separate buffer. If an error occurs, the bad buffer returns to software control. Software clears the error bits in the buffer descriptor, resets the addresses to point to the start of the data buffer, and retransfers the data. Handling a USB BUF_ERR on an OUT endpoint is complicated. The process is similar to the process required for BUF_ERRs on Buffer-per-IRP mode as described below.

Error Handling in Buffer-Per-IRP Mode

Error handling is more complex in buffer-per-IRP mode. There are four separate error/direction combinations, and each condition is discussed below.

Handling USB BUF_ERR on an IN Endpoint With a SmartDMA Channel

The BUF_ERR bit is set in the USB endpoint control register (xEPCTL) when a buffer underun occurs—the endpoint requests data, but the SmartDMA controller either cannot distribute the data quickly enough, or has no available data. This error always occurs when an IN endpoint is activated with SmartDMA channel and there is no data (even if the SmartDMA channel is not turned on). The interrupt should be masked until data is available. Assuming data is available when this error occurs, perform the following steps:

1. Read the SmartDMA channel current transfer address and compare it to the data buffer address. The difference is the amount of data already transmitted.
2. Find the offset from the data buffer base address to the start of the bad packet by removing the low-order bits (those that are smaller than the packet size) from the amount of data transferred.
3. Add the packet-start offset to the buffer base-address and store the sum in the Address word of the buffer descriptor. The buffer descriptor now points to the start of the packet that needs to be resent.
4. Adjust the byte count word to adjust for the shortened buffer length.
5. Turn the SmartDMA channel back on. The host sends a new IN token, and the packet is re-sent.

Handling USB OTHER_ERR on an IN Endpoint with a SmartDMA Channel

The OTHER_ERR bit in the xEPCTL register is set when the host responds with a NAK to the most recently transmitted packet. The SmartDMA channel transmit pointer must be backed up to the start of the NAK-response packet. Because part of a new packet might be moved into the FIFO, this might involve backing up more than one packet size. The following is the recovery process:

1. Read the SmartDMA channel current transfer address and compare it to the data buffer address. The difference is the amount of data already transmitted.
2. Reduce the amount of data transferred by one packet size, and then perform an AND operation to the amount transferred with the packet size. The result is the offset from the start of the data buffer to the start of the NAK-response packet.
3. Add the packet-start offset to the buffer base-address and store the sum in the Address word of the buffer descriptor. The buffer descriptor now points to the start of the packet that needs to be resent.
4. Adjust the Byte Count word to adjust for the shortened buffer length.
5. Turn the SmartDMA channel back on. The host will send a new IN token, and the packet is resent.

Handling USB BUF_ERR on an OUT Endpoint with a SmartDMA Channel

The BUF_ERR bit in the xEPCTL register is set when the SmartDMA channel is unable to move data from the USB FIFO fast enough. The result is the FIFO contains part of the packet, which the host has acknowledged and cannot resend, and contains part of an overrun packet, which the host can resend. The error handler must retrieve the rest of the acknowledged packet and update the SmartDMA pointers so that the host can resend the bad packet.

1. Read the SmartDMA current receive address and compare it to the data buffer address. The difference is the amount of data already received.
2. If more than a packet-size of data is received, perform a modulo on the received amount with the packet size. This value is the amount of the acknowledged packet that is written to memory. In a loop, the rest of the data must be read from the USB FIFO and written into memory.
3. Update the SmartDMA channel descriptor to point to the start of the packet that was in transmit when the error occurred. If data was read from the FIFO in step 2, then the pointer does this already.
4. Turn the SmartDMA channel back on. The host resends the interrupted packet.

Handling USB OTHER_ERR on an OUT Endpoint with a SmartDMA Channel

The OTHER_ERR bit is set when the device receives a bad packet. The device immediately issues a NAK to the packet and the host automatically resends it. The error handler needs to rewind the SmartDMA channel so that the re-sent packet overwrites the bad packet data. This is done as follows:

1. Read the SmartDMA channel current receive address and compare it to the data buffer address. The difference is the amount of data already received.
2. Reduce the amount of data received by a packet size. This is the offset of the start of the bad packet.
3. Add the offset to the start of the SmartDMA channel data buffer and assign the sum to the Address word of the ring descriptor.
4. Adjust the Byte Count field to account for the shortened buffer.
5. Restart the SmartDMA channel and wait for the host to resend.

EXAMPLE CODE

Refer to Appendix A for an example of code that shows error recovery procedures for USB with the SmartDMA controller. The code is not a fully functioning USB driver and requires a HandleOtherInt() function to process transactions (e.g., initialization, configuration) over the control endpoint.

Example Code

The example code is a simple loopback on endpoints C and D. Data can also be sent to the device on endpoint A, but the data is discarded. This code provides an example of error-handling procedures for USB with the SmartDMA controller. The code is not a fully functional USB driver. This code must be combined with EPD CodeKits CK0005xx or CK0012xx to implement a fully functioning USB system using the SmartDMA controller. To access the CodeKits, refer to the following link on the AMD web site: www.amd.com/products/lpd/codekits/downloads.html.

```
#include "Am186CC.h"
#include "usb.h"

WORD GetDiff(WORD num1, WORD num2);
WORD GetLinearAddress(void far *fptr, DWORD *address);
WORD FindBuffer(SDMADescriptorFormat *ring_buffer, WORD active, WORD cbd);

void ConfigureUSB();
void ConfigureSDMA();
void UsbHandleInt();
void UsbProcessInt();
void HandleErrorD();
void HandleErrorA();
void HandleReceivedPacketA();
void HandleErrorC();
void HandleReceivedPacketC();
void HandleOtherInterrupt();
void SDMAHandleInt();
void SDMAProcessInt();
typedef struct SmartDMABufferDescriptorFormat
{
      WORD     Low_address;   // Low order bits of address field
      WORD     Config;        // Status and configuration
      WORD     Size;          // Receive buffer size in bytes
      WORD     Message_count; // Frame length in bytes
} SDMADescriptorFormat;

#define DESCRIPTORS 16          /* Number of descriptors in ring buffer */
#define PACKET_SIZE 64          /* Size of a USB packet */
#define A_BUF_SIZE  PACKET_SIZE /* Size of endpoint A’s buffers */
#define C_BUF_SIZE  512         /* Size of endpoint C’s buffers */

char data_buf_a[DESCRIPTORS][A_BUF_SIZE]; /* Data buffers for endpoint A */

volatile SDMADescriptorFormat far ring_buffer_a[DESCRIPTORS]; /* These are
      ** the rings of format descriptors. Each descriptor contains the
      ** status for a buffer. Must be volatile since the hardware has
      ** access to it. */
char data_buf_c[DESCRIPTORS][C_BUF_SIZE]; /* Data buffers for endpoint C */
volatile SDMADescriptorFormat far ring_buffer_c[DESCRIPTORS]; /* These are
      ** the rings of format descriptors. Each descriptor contains the
      ** status for a buffer. Must be volatile since the hardware has
      ** access to it. */
/* endpoint D has no data buffers - it will use either data_buf_c or
      ** data_buf_a. */
volatile SDMADescriptorFormat far ring_buffer_d[DESCRIPTORS]; /* These are
      ** the rings of format descriptors. Each descriptor contains the
      ** status for a buffer. Must be volatile since the hardware has
      ** access to it. */
WORD a_current; /* Number of the buffer currently being used by this */
WORD c_current; /* SmartDMA channel, or the number of the most recently */
WORD d_current; /* Used channel. */

void main(void)
{
      ConfigureUSB();
      ConfigureSDMA();
      /* Loop forever */
      for (;;)
            ;
}

/*****************************************************************************
NAME
      GetDiff - Return the absolute difference between two numbers
DESCRIPTION
      Configures endpoint
*****************************************************************************/
WORD GetDiff(WORD num1, WORD num2)
{
      if (num1 > num2)
            return(num1 - num2);
      else
            return(num2 - num1);
}

/*****************************************************************************
NAME
      GetLinearAddress - Get the linear address of a pointer
DESCRIPTION
*****************************************************************************/
WORD GetLinearAddress(void far *fptr, DWORD *address)
{
      DWORD seg, offset;
      seg = (unsigned) ((DWORD)(fptr) >> 16);
      offset = (unsigned) ((DWORD)(fptr) & 0xffff);
      seg <<= 4;
      *address = seg + offset;
      return 0;
}

/*****************************************************************************
NAME
      FindBuffer - Search for an unused or used buffer
DESCRIPTION
*****************************************************************************/
WORD FindBuffer(SDMADescriptorFormat *ring_buffer, WORD active, WORD cbd)
{
      WORD found, i;
      found = 0;
      for (i = cbd; (i < DESCRIPTORS) && (!found); i++)
      {
            if ((ring_buffer[i].Config & SDBUF_OWN) == active)
                  found = i + 1;
      }
      for (i = 0; (i < cbd) && (!found); i++)
      {
            if ((ring_buffer[i].Config & SDBUF_OWN)== active)
                  found = i + 1;
      }
      return found;
}
/*****************************************************************************
NAME
      ConfigureUSB - Configures USB endpoint A, C, and D
DESCRIPTION
      Configures each endpoint’s definition registers
*****************************************************************************/
void ConfigureUSB()
{
      /* Turn off USB interrupts. */
      outpw(UIMASK1, 0);
      outpw(UIMASK2, 0);

      /* Set endpoint A as a BULK/OUT endpoint with "buffer per packet
      ** IRP" SmartDMA. */
      /* Definition register 1: Set the endpoint number to B, the configuration,
      ** interface, and alternate setting to 0, the direction to IN, and
      ** the type to BULK. */
      outpw(UAEPDEF1, UEPDEF1_EP_NUM_A | UEPDEF1_EP_CFG_0 |
            UEPDEF1_EP_INT_0 | UEPDEF1_EP_ASET_0 |
            UEPDEF1_EP_DIR_OUT | UEPDEF1_EP_TYPE_BULK);
      /* Definition register 2: 64 byte FIFO (packet size), and 64 byte
      ** packets */
      outpw(UAEPDEF2, UEPDEF2_FIFO_SIZE_16 | UEPDEF2_EP_MX_PCT_64);
      /* Definition register 3: SmartDMA with packet status,
      ** and an interrupt is generated on FULL_PKT or SHRT_PKT and a
      ** stop interrupt on BUFF_ERR  or OTHER_ERR. The bits are shifted 8 to
      ** the left to move  them from the "stop" positions to the "status" positions. */
      outpw(UAEDEF3, UEPDEF3_MODE_SDMA_STAT | ((UEPCTL_FULL_PKT |
            UEPCTL_SHOR_PKT) << 8) | UEPCTL_BUF_ERR |
            UEPCTL_OTHER_ERR);

      /* Set endpoint C as a BULK/OUT endpoint with "buffer per IRP" SmartDMA. */
      /* Definition register 1: Set the endpoint number to C, the configuration,
      ** interface, and alternate setting to 0, the direction to OUT, and the type to
      ** BULK. */
      outpw(UCEPDEF1, UEPDEF1_EP_NUM_C | UEPDEF1_EP_CFG_0 |
            UEPDEF1_EP_INT_0 | UEPDEF1_EP_ASET_0 |
            UEPDEF1_EP_DIR_OUT | UEPDEF1_EP_TYPE_BULK);
      /* Definition register 2: 64-byte FIFO (packet size), and 64 byte packets */
      outpw(UCEPDEF2, UEPDEF2_FIFO_SIZE_64 |UEPDEF2_EP_MX_PCT_64);
      /* Definition register 3: No packet status in the OUT direction,
      ** and a stop interrupt is generated on SHRT_PKT, BUFF_ERR, or OTHER_ERR. */
      outpw(UCEDEF3, UEPDEF3_MODE_SDMA | UEPCTL_FULL_PKT |
            UEPCTL_BUF_ERR | UEPCTL_OTHER_ERR);
      /* Set endpoint D as a BULK/IN endpoint with "buffer per IRP" SmartDMA. */
      /* Definition register 1: Set the endpoint number to C, the configuration,
      ** interface, and alternate setting to 0, the direction to IN, and the type to
      ** BULK. */
      outpw(UDEPDEF1, UEPDEF1_EP_NUM_D | UEPDEF1_EP_CFG_0 |
            UEPDEF1_EP_INT_0 | UEPDEF1_EP_ASET_0 |
            UEPDEF1_EP_DIR_IN | UEPDEF1_EP_TYPE_BULK);
      /* Definition register 2: 32-byte FIFO (packet size), and 64-byte packets */
      outpw(UDEPDEF2, UEPDEF2_FIFO_SIZE_32 | UEPDEF2_EP_MX_PCT_64);
      /* Definition register 3: No packet status in the OUT direction,
      ** and a stop interrupt is generated on SHRT_PKT, BUFF_ERR, or OTHER_ERR. */
      outpw(UDEDEF3, UEPDEF3_MODE_SDMA | UEPCTL_BUF_ERR |
            UEPCTL_OTHER_ERR);

      /* Set up the USB interrupt channel and vector. */
      {
            unsigned long far *VectorTable;
            /* Address of handler offset value */
            VectorTable = (unsigned long far *) (ITYPE_USB << 2);
            *VectorTable = (unsigned long) USBHandleInt;
      }
      outpw(CH2CON, CHCON_SRC_INTERNAL | CHCON_LTM | CHCON_PR6);
      /* Turn on the receive interrupts. */
      outpw(UIMASK1, UIMASK1_C_EP_ACT | UIMASK1_A_EP_STATINT |
            UIMASK1_A_EP_ACT | UIMASK1_OTHER_INT
            UIMASK1_CNT_EP_NEW |UIMASK1_CNT_EP_ACT);
}

/*****************************************************************************
NAME
ConfigureSDMA - Configures SmartDMA channels 2 and 3
DESCRIPTION
Configures the SmartDMA channels to work with endpoints A, C, and D
*****************************************************************************/
void ConfigureSDMA()
{
      DWORD address;
      WORD i;
      /* Set up SDMA Channel 2. */
      /* SDMA Control register: Receive Set Own, medium priority, USB
      ** as requesting source. */
      outpw(SD2CON, SDCON_RXSO | SDCON_P_MED | SDCON_DEL_USB);
      GetLinearAddress(ring_buffer_a, &address);
      outpw(SD2RRCAL, ((WORD) address & 0xfff0)| SDRRCAL_RRC_16);
      outpw(SD2RRAH, (WORD) ((address >> 16) & 0x000f));

      for (i = 0; i < DESCRIPTORS; i++)
      {
            GetLinearAddress(data_buf_a[i], &address);
            ring_buffer_a[i].Low_address = (WORD) address & 0xffff;
            ring_buffer_a[i].Low_address =  SDBUF_OWN | 0x0300 |
                  ((WORD) (address >> 16) & 0x000f);
            ring_buffer_a[i].Size = (~A_BUF_SIZE + 1) & 0x7FFF;
            ring_buffer_a[i].Message_count = 0;
      }
      a_current = 0;
      /* Set up SDMA Channel 3. */
      /* SDMA Control register: Transmit terminal count interrupt,
      ** medium priority, USB as requesting source. */
      outpw(SD3CON, SDCON_TTCI | SDCON_TXSO | SDCON_RXSO | SDCON_P_MED |
            SDCON_DEL_USB);
      GetLinearAddress(ring_buffer_c, &address);
      outpw(SD3RRCAL, ((WORD) address & 0xfff0)| SDRRCAL_RRC_16);
      outpw(SD3RRAH, (WORD) ((address >> 16) & 0x000f));

      /* Set up the receive ring descriptor to point to the available
      ** data buffers. */
      for (i = 0; i < DESCRIPTORS; i++)
      {
            GetLinearAddress(data_buf_c[i], &address);
            ring_buffer_c[i].Low_address = (WORD) address & 0xffff;
            ring_buffer_c[i].Low_address =  SDBUF_OWN | 0x0300 |
                  ((WORD) (address >> 16) & 0x000f);
            ring_buffer_c[i].Size = (~C_BUF_SIZE + 1) & 0x7FFF;
            ring_buffer_c[i].Message_count = 0;
      }
      c_current = 0;

      GetLinearAddress(ring_buffer_d, &address);
      outpw(SD3TRCAL, ((WORD) address & 0xfff0)| SDTRCAL_TRC_16);
      outpw(SD3TRAH, (WORD) ((address >> 16) & 0x000f));
      /* Set up the transmit ring descriptors to not point anywhere. Leave
      ** them under software control so we can adjust them after we receive
      ** data. */
      for (i = 0; i < DESCRIPTORS; i++)
      {
            ring_buffer_d[i].Low_address = 0;
            ring_buffer_d[i].Low_address = 0x0300;
            ring_buffer_c[i].Size = 0;
            ring_buffer_c[i].Message_count = 0;
      }
      /* Set up the SDMA interrupt channels and vectors. */
      {
            unsigned long  far *VectorTable;
            /* Address of handler offset value */
            VectorTable = (unsigned long far *) (ITYPE_SDMA3 << 2);
            *VectorTable = (unsigned long) SDMAHandleInt;
      }
      outpw(CH7CON, CHCON_PR6);
}


/*****************************************************************************
NAME
      UsbHandleInt - USB Interrupt handler
DESCRIPTION
      Calls the USB interrupt processor and cleans up the end-of-interrupt
      register afterwards
*****************************************************************************/
void interrupt UsbHandleInt()
{
      UsbProcessInt();
      outpw(EOI, EOITYPE_USB);
}

/*****************************************************************************
NAME
      UsbProcessInt - Determine interrupt cause and finish interrupt
DESCRIPTION
      This function loops, checking to see what condition caused the USB
      interrupt, until all sources are handled. Depending on the interrupt
      source, it will usually shut down a SDMA channel and mask out an
      endpoint interrupt while it uses a subfunction to handle the source.
*****************************************************************************/
void UsbProcessInt()
{
      /* First determine which endpoint is causing the interrupt. */
      volatile WORD status;
      while ((status = inpw(UISTAT1)) != 0)
      {
            /* Check for transmit errors first. */
            if (status & UISTAT1_D_EP_ACT)
            {
                  WORD uimask, sdcon;
                  /* Stop the DMA while we figure out the interrupt source. */
                  uimask = inpw(UIMASK1);
                  sdcon = inpw(SD3CON);
                  outpw(UIMASK1, uimask & ~UISTAT1_D_EP_ACT);
                  outpw(SD3CON,  sdcon & ~SDCON_TXST);
                  /* Since this is a transmit line, we've programmed SmartDMA to
                  ** interrupt after completely sending a packet. The USB only
                  ** interrupts if there's been an error. */
                  HandleErrorD();
                  /* Now that we've handled the interrupt, clear the
                  ** act_req bit, turn interrupts back on, and resume. */
                  outpw(UDEPCTL, inpw(UDEPCTL) &~ UEPCTL_ACT_REQ);
                  outpw(UIMASK1, uimask);
                  outpw(SD3CON, sdcon);
            }
            /* Then check for actions on the A endpoint. */
            else if (status & UISTAT_A_EP_ACT)
            {
                  WORD a_status, uimask, sdcon;
                  /* Stop the DMA while we figure out the interrupt source. */
                  uimask = inpw(UIMASK1);
                  sdcon = inpw(SD2CON);
                  outpw(UIMASK1, uimask & ~UISTAT1_A_EP_ACT);
                  outpw(SD2CON,  sdcon & ~SDCON_RXST);
                  a_status = inpw(UAEPCTL);
                        if ((a_status & UEPCTL_BUF_ERR)||(a_status (UEPCTL_OTHER_ERR)
                        HandleErrA();
                  /* Now that we've handled the interrupt, clear the
                  ** act_req bit, turn interrupts back on, and resume. */
                  outpw(UAEPCTL, inpw(UAEPCTL) &~ UEPCTL_ACT_REQ);
                  outpw(UIMASK1, uimask);
                  outpw(SD2CON, sdcon);
            }
            else if (status & UISTAT_A_STATINT)
            {
                  /* The SmartDMA places each new packet in its own buffer, so
                  ** there’s no need to stop on that. We can keep receiving
                  ** data while we handle each packet. */
                  HandleReceivedPacketA();
            }
            else if (status & UISTAT_C_EP_ACT)
            {
                  WORD c_status, uimask, sdcon;
                  /* Stop the DMA while we figure out the interrupt source. */
                  uimask = inpw(UIMASK1);
                  sdcon = inpw(SD3CON);
                  outpw(UIMASK1, uimask & ~UISTAT1_C_EP_ACT);
                  outpw(SD3CON,  sdcon & ~SDCON_RXST);
                  c_status = inpw(UCEPCTL);
                  /* There can either be an error condition, or a short packet
                  ** condition on this endpoint. Short packet means the host is
                  ** done sending the IRP and we need to handle the packet. */
                  if ((c_status & UEPCTL_BUF_ERR)|| (c_status & (UEPCTL_OTHER_ERR))
                        HandleErrC();
                  else if (c_status & (UEPCTL_SHOR_PKT) HandleReceivedPacketC();
                  /* Now that we’ve handled the interrupt, clear the
                  ** act_req bit, turn interrupts back on, and resume. */
                  outpw(UCEPCTL, inpw(UCEPCTL) &~ UEPCTL_ACT_REQ);
                  outpw(UIMASK1, uimask);
                  outpw(SD3CON, sdcon);
            }
            else
                  /* Must be a control endpoint or general USB status interrupt. */
                  HandleOtherInterrupt();
      }
}

/****************************************************************************
NAME
      HandleErrorD - Handle an error on the D endpoint
DESCRIPTION
      This function handles an error on the D endpoint. If there is a
      BUFFER (underrun) error, it retransmits the current packet.
      If there is an OTHER (NAK) error, it retransmits the previous packet.
****************************************************************************/
void HandleErrorD()
{
      WORD current_address, current_ring, length;
      WORD error_cause = inpw(UDEPCTL);
      current_ring = inpw(SD3CBD) & 0x00ff;
      current_address = inpw(SD3TCAD);
      length = GetDiff(current_address, ring_buffer_d[current_ring].Low_address);
      /* First, determine the source of the error. */
      if (error_cause & UEPCTL_BUF_ERR)
      {
            /* BUFF_ERR indicates an error in the current packet. We need to
            ** point the SDMA to the start of the current packet and resend it. */
            /* Each packet is PACKET_SIZE bytes, so chopping off the lower bits will
            ** give us the offset of the packet start. Add that to the
            ** beginning of the buffer. */
            ring_buffer_d[current_ring].Low_address+= length & ~(PACKET_SIZE - 1);
            /* The buffer length has decreased by the amount of data
            ** already transmitted. */
            ring_buffer_d[current_ring].Size -= length & (PACKET_SIZE - 1);
            /* Flush the FIFO and clear the error bit. */
            outpw(UDEPCTL, inpw(UAEPCTL) & ~(UEPCTL_NOT_FLUSH | UEPCTL_BUF_ERR));
            }
      else if (error_cause & UEPCTL_OTHER_ERR)
      {
            /* OTHER_ERR indicates the previous packet was NAK’ed, so we need to
            ** point the SDMA to the previous packet and resend it. */
            if (length < PACKET_SIZE)
                  /* No previous packet. We shouldn’t get this, but set the
                  ** length to PACKET_SIZE just to be safe. */
                  length = PACKET_SIZE;
            length -= PACKET_SIZE;  /* Back up a packet length. */
            /* Each packet is PACKET_SIZE bytes, so chopping off the lower bits will
            ** give us the offset of the packet start. Add that to the
            ** beginning of the buffer. */
            ring_buffer_d[current_ring].Low_address+= length & PACKET_SIZE;
            /* The buffer length has to be decreased by the amount of data
            ** already transmitted. */
            ring_buffer_d[current_ring].Size -= length & PACKET_SIZE;
            /* Flush the FIFO and clear the error bit. */
            outpw(UDEPCTL, inpw(UAEPCTL) & ~(UEPCTL_NOT_FLUSH | UEPCTL_OTHER_ERR));
      }
      else
      {
            printf("unknown interrupt source");
      }
}

/*****************************************************************************
NAME
      HandleErrorA - Handle an error on the A endpoint
DESCRIPTION
      This function recovers the data for any buffer with partially
      written data in it, and discards any buffer with bad data in it.
*****************************************************************************/
void HandleErrorA()
{
      ring_buffer_a[a_current].Config &= ~SDBUF_OWN;
      DWORD address;
      WORD low_address, status;
      GetLinearAddress(data_buf_a[a_current], &address);
      low_addres = (WORD) address & 0xffff;
      status = inpw(UAEPCTL);
      if (status & UEPCTL_OTHER_ERR)
      {
            /* The packet was rejected. Discard it and wait for the host
            ** to retransmit. */
            ring_buffer_a[a_current].Size = (~A_BUF_SIZE + 1) & 0x7FFF;
            ring_buffer_a[a_current].Low_address = low_address;
            ring_buffer_a[a_current].Message_count = 0;
            ring_buffer_a[a_current].Config = (WORD) ((address >> 16) &
                   0x000f)) | SDBUF_OWN | 0x0200;
      }
      else
      {
            WORD length, packet_size, i;
            char * data_buf = data_buf_a[a_current];
            /* Was the previous packet full-length, or short? We know that
            ** full-length packets have a total of packet-size bytes in
            ** them, and we can read the PACKET_SIZE register to get the
            ** size of shorter packets. */
            if (status & UEPCTL_SHOR_PKT)
                  packet_size = inpw(UAEPSIZ);
            else
                  packet_size = PACKET_SIZE;
            length = GetDiff(inpw(SD2CRAD), ring_buffer_a[a_current].Low_address);
            if (length > packet_size)
            {
                  /* If the amount of data written from the FIFO is longer than
                  ** the amount of data originally in the FIFO, we have a real
                  ** problem. */
                  printf("error\n"); return;
            }
            /* We need to read the unread data from the FIFO. It starts
            ** at the length offset into the FIFO, and continues to the
            ** packet size. */
            for (i = length; i < packet_size; i++)
            {
                  /* Read the data from the FIFO and write it to memory. */
                  data_buf[length + i] = inpw(UAEPDAT);
            }
            /* Update the message count and clear the error bit. */
            ring_buffer_a[a_current].Message_count = packet_size;
            ring_buffer_a[a_current].Config = (WORD) ((address >>16) & 0x000f)) |
                   0x0200;
            /* We have now received a packet. Invoke the packet handler
            ** routine. */
            HandleReceivedPacketA();
      }
      /* Clear the error bits and flush the FIFO. */
      outpw(UAEPCTL, inpw(UAEPCTL) & ~(UEPCTL_NOT_FLUSH | UEPCTL_BUF_ERR
            |UEPCTL_OTHER_ERR));
}
/*****************************************************************************
NAME
      HandleReceivedPacketA - Handle a successful receive on the A endpoint
DESCRIPTION
      This function handles packets on the A endpoint by dropping them to
      the floor and resetting the most recently used buffer descriptor.
*****************************************************************************/
void HandleReceivedPacketA()
{
      DWORD address;
      /* Get the original address of this data buffer. */
      GetLinearAddress(data_buf_a[a_current], &address);
      /* Reset the buffer. */
      ring_buffer_a[a_current].Size = (~PACKET_SIZE + 1) & 0x7FFF;
      ring_buffer_a[a_current].Low_address = (WORD) address & 0xffff;
      ring_buffer_a[a_current].Config = (WORD) ((address >> 16) & 0x000f)) | SDBUF_OWN
            | 0x0200;
      a_current = (outpw(SD2CBD) & 0xff00) >> 8;
}
```

```
/****************************************************************************
NAME
      HandleErrorC - Handle an error on the C endpoint
DESCRIPTION
      This function recovers from a buffer error on the C endpoint.
      If a buffer overrun caused the error, the function pulls any
      data from a previously ACK'ed packet from the FIFO and writes it to
      memory. It then updates the SDMA descriptor to point to after the end
      of the newly written packet and prepares for the host to retransmit
      the NAK'ed packet. If the error was caused by the device NAK'ing a
      packet, the function rewinds the SDMA descriptor to point to the start
      of the NAK'ed packet. The host will retransmit normally.
****************************************************************************/
void HandleErrorC()
{
      DWORD address; WORD low_address, length, bytes_left;
      WORD error_cause = inpw(UDEPCTL);
      GetLinearAddress(data_buf_c[c_current], &address);
      low_address = inpw(SD3RCAD);
      length = GetDiff(low_address, ring_buffer_c[c_current].Low_address);
      /* Check to see what the error condition is. */
      if (error_cause & UEPCTL_BUF_ERR)
      {
            /* BUFF_ERR indicates an overflow in the current packet. Part of
            ** the previous packet might be in the buffer, so we need to
            ** pull that out and store it in the data buffer. Then we need
            ** to point the DMA to the start of the current buffer, flush the
            ** buffer, and continue. */
            if (length > PACKET_SIZE)
            {
                  int bytes_left, i;
                  char * data_buf = data_buf_c[c_current];
                  /* Determine the number of bytes left from the previous packet. */
                  bytes_left = PACKET_SIZE - (length % PACKET_SIZE);
                  /* Read the bytes left from the previous packet. */
                  for (i = 0; i < bytes_left; i++) data_buf[length + i] =
                        inpw(UCEPDAT);
                  /* Update the length to point to past the end of the packet that
                  ** was just read from the data FIFO. */
                  length = (length & PACKET_SIZE) + PACKET_SIZE;
            }
            /* Point the data to the start of the interrupted packet. */
            ring_buffer_c[c_current].Low_address += length & PACKET_SIZE;
            /* The buffer length has to be decreased by the amount of data
            ** already received. */
            ring_buffer_c[c_current].Size -= length & PACKET_SIZE;
            outpw(UCEPCTL, inpw(UCEPCTL) & ~(UEPCTL_NOT_FLUSH | UEPCTL_BUF_ERR));
      }
      else if (error_cause & UEPCTL_OTHER_ERR)
      {
            /* OTHER_ERR indicates the most recently received packet was NAK'ed,
            ** so we need to rewind the SDMA so that it will overwrite the
            ** previous packet. */
            if (length >= PACKET_SIZE) length -= PACKET_SIZE;
            /* Point the write pointer to the start of the NAK'd packet. */
            ring_buffer_c[c_current].Low_address += length & PACKET_SIZE;
            /* The buffer length has to be decreased by the amount of data
            ** already received. */
            ring_buffer_c[c_current].Size -= length & PACKET_SIZE;
            outpw(UCEPCTL, inpw(UCEPCTL) & ~(UEPCTL_NOT_FLUSH |
            UEPCTL_OTHER_ERR));
      }
      else
      {
            printf("unknown interrupt source");
      }
}

/***************************************************************************
NAME
      HandleReceivedPacketC - Handle a successful receive on the C endpoint
DESCRIPTION
      An entire buffer has been received, so we need to queue it up for
      transmit on endpoint D. The first step is to find an unused buffer
      descriptor in the endpoint D ring. (The handler will continue to be
      called until a buffer is available.) We need to set the D descriptor
      to point to the start of the "current" C data buffer. If the D channel
      is not currently transmitting, it is turned on so it can transmit at the
      next request of the host. Once the buffer has been transferred to D’s
      control, we need to find another receive buffer.
***************************************************************************/
void HandleReceivedPacketC()
{
      /* Short packets indicate the end of an IRP. Process the buffer
      ** by passing it to the transmit channel for loopback. */
      DWORD address; WORD cbd, found;
      /* First find an unused buffer descriptor on endpoint D. */
      cbd = inpw(SD3CBD) & 0x00ff; found = FindBuffer(ring_buffer_d, cbd, 0);
      if (found)
      {
            /* Point the D descriptor to the recently received buffer. */
            WORD length;
            GetLinearAddress(data_buf_c[c_current], &address);
            ring_buffer_d[found - 1].Low_address = (WORD) address & 0xffff;
            ring_buffer_d[found - 1].Config |= (WORD) ((address >> 16) & 0x000f)) |
                  0x0200;
            /* The difference between the data buffer start and the location
            ** of the last SDMA write is the length of the buffer. We could
            ** also use the 4th word of the descriptor if we were writing
            ** status, which we aren’t. */
            length = GetDiff(inpw(SD3CRAD), ring_buffer_d[found - 1].Low_address);
            ring_buffer_d[found -1].Size = 0x8000 | (~length + 1);
            ring_buffer_d[found - 1].Config |= SDBUF_OWN;
            /* Check to see if the SDMA transmitter is active. If it isn’t,
            ** make it active and turn on the interrupts. Otherwise, let the
            ** SDMA transfer interrupt handle the buffer. */
            if ((SD3CON0 & SDCON_TXST) == 0)
            {
                  /* Point the CBD to the found buffer. */
                  outpw(SD3CBD, inpw(SD3CBD) | found - 1);
                  /* Turn on SmartDMA transmission. */
                  outpw(SD3CON, inpw(SD3CON) | SDCON_TXST);
                  /* Clear the USB of any errors. */
                  outpw(UDEPCTL, inpw(UDEPCTL) & ~UEPCTL_OTHER_ERR);
                  /* Turn on USB interrupts. */
                  outpw(UIMASK1, inpw(UIMASK1) | UISTAT1_D_EP_ACT);
            }
            /* Find the next empty receive buffer descriptor. */
            found = FindBuffer(ring_buffer_c, c_current, SDBUF_OWN);
            if (found)
            {
            /* Return the current buffer to the available pool. */
            ring_buffer_c[c_current].Size = (~C_BUF_SIZE + 1) & 0x7FFF;
            ring_buffer_c[c_current].Low_address = (WORD) address & 0xffff;
            ring_buffer_c[c_current].Config = (WORD) ((address >>
                   16) & 0x000f)) | SDBUF_OWN | 0x0200;
            /* Point the SDMA to this buffer so that the next OUT token
            ** will write to it. */
            c_current = found - 1;
            outpw(SD3CBD, inpw(SD3CDB) | (c_current << 8));
            /* Clear the short packet bit now that the interrupt is
            ** fully handled. */
            outpw(UCEPCTL, inpw(UCEPCTL) & ~UEPCTL_SHOR_PKT);
            }
      }
}
/*****************************************************************************
NAME
      HandleOtherInterrupt - Handle non-DMA interrupts
DESCRIPTION
      This is a placeholder function for full-featured USB interrupt handler.
*****************************************************************************/
void HandleOtherInterrupt()
{
      /* This function would handle Control endpoint messages
      ** and other standard USB interrupts. For a detailed
      ** example of a full-featured USB interrupt handler,
      ** study CodeKits CK0005xx or CK0012xx available on the EPD
      ** website. */
}

/*****************************************************************************
NAME
      SDMAHandleInt - SmartDMA interrupt
DESCRIPTION
      This function should be activated when SDMA channel 3 generates a
      transmit terminal count interrupt. It calls the interrupt processor
      and writes to the end of the interrupt register.
*****************************************************************************/
void interrupt SDMAHandleInt()
{
      SDMAProcessInt();
      outpw(EOI, EOITYPE_SDMA3);
}

/*****************************************************************************
NAME
      SDMAProcessInt - Handle a successful transmit on the D endpoint
DESCRIPTION
      This function is called when an entire buffer is transmitted. It
      masks the USB interrupt to avoid being repeatedly called on buffer
      underrun and turns off the SDMA transmit channel. It then checks
      to see if there are any other buffers that need to be transmitted.
      If there are, it points the CBD at the buffer and turns the channel
      and interrupts back on. The transmit buffers are prepared by the C
      channel receive function, HandleReceivedPacketC().
*****************************************************************************/
void SDMAProcessInt()
{
      WORD found, cbd;
      /* Check for interrupt source. */
      if ((inpw(SD3STAT) & SDSTAT_TTC) == 0)
      {
            /* TTC is the only interrupt we should get. */
            printf("unknown interrupt\n");
            return;
      }
      else
            /* Clear the interrupt. */
            outpw(SD3STAT, 0);
      /* Turn off USB interrupts and SmartDMA transmission. */
      outpw(UIMASK1, inpw(UIMASK1) & ~UISTAT1_D_EP_ACT);
      outpw(SD3CON, inpw(SD3CON) & ~SDCON_TXST);
      /* Check for any additional buffers that need to be
      ** transmitted. These buffers have already been prepared by the
      ** receiver function. */
      cbd = inpw(SD3CBD) & 0x00ff;
      found = FindBuffer(ring_buffer_d, cbd, SDBUF_OWN);
      if (found)
      {
            /* Point the CBD to the found buffer. */
            outpw(SD3CBD, inpw(SD3CBD) | found - 1);
            /* Turn on SmartDMA transmission. */
            outpw(SD3CON, inpw(SD3CON) | SDCON_TXST);
            /* Clear the USB of any errors. */
            outpw(UDEPCTL, inpw(UDEPCTL) & ~UEPCTL_OTHER_ERR);
            /* Turn on USB interrupts. */
            outpw(UIMASK1, inpw(UIMASK1) | UISTAT1_D_EP_ACT);
      }
}
```