

```

//-----
// Copyright (c) 1999 Cypress Semiconductor, Inc. All rights reserved
//-----
#include <ezusb.h>
#include <ezregs.h>
#include <scsi.h>
#include "atapi.h"
#if FLASH
#include "flashcrd.h"
#else
    #if GPIF
    #define AGGRESSIVELY_USE_TNG_FEATURES 1
    #include "gpif.h"
    #include "tng.h"
    #else
    #include "fastxfer.h"
    #endif
#endif

bit waitForInBuffer();
bit ideReadCommand(bit verify);
bit ideWriteCommand();
bit ideReadCommandTest();
bit ideWriteCommandTest();
void IDEnop();

// For 2200 series make sure to locate this in a different RAM space than the code.
// For example:
// echo fw_gpif.obj, reset.obj, ide_gpif.obj, scsigpif.obj, usbjmbptb.obj, dscr.obj, per_gpif.obj, gpif.obj, >
tmp.rsp
// echo %EZTARGET%\lib\EZUSB.lib >> tmp.rsp
// echo TO gpif RAMSIZE(256) CODE(00h,?CO?IDE(1800h),?CO?PERIPH) XDATA(1a00h) IXREF >> tmp.rsp
// bl51 @tmp.rsp
// See DMA chapter in the manual for more details.
const char code senseInvalidFieldInCDB[] = {0x70, 0x00, 0x05, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x24, 0x00, 0x00, 0x00, 0x00, 0x00};
const char code senseOk[] = {0x70, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
const char code senseNoMedia[] = {0x70, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00, 0x00, 0x12, 0x00, 0x00, 0x00,
0x00, 0x3a, 0x00, 0x00, 0x00, 0x00, 0x00};
const char code senseMediaChanged[] = {0x70, 0x00, 0x06, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x28, 0x00, 0x00, 0x00, 0x00, 0x00};
char code * sensePtr;

bit generalIDEInCommand()
{
    BYTE cmd;
    bit status;

    cmd = OUT2BUF[0xf];

    switch (cmd)
    {
        // Minimum processing for a case in this switch statement:
        //
        // EPIO[OUT2BUF_ID].bytes = 0; // relinquish control of the bulk buffer occupied by the CBW
        // sensePtr = senseInvalidFieldInCDB; // set the sense pointer for the current command
        // dataTransferLen = updated value
        // return(USBS_FAILED); // return PASSED or FAILED
        case INQUIRY:
        {
            BYTE packetLen = min(dataTransferLen, sizeof(SCSIInquiryData));

            EPIO[OUT2BUF_ID].bytes = 0; // relinquish control of the bulk buffer occupied by the CBW
            // Send out our stored inquiry data
            waitForInBuffer();
            mymemmovexx(IN2BUF, SCSIInquiryData_, packetLen);
            IN2BC = packetLen;
            dataTransferLen -= packetLen;
            sensePtr = senseOk;
            return(USBS_PASSED);
        }

        case READ_10:
            sensePtr = senseOk;
            checkForMedia();
            if (sensePtr == senseOk)
                return(ideReadCommand()); // ideReadCommand sets the OUT2BC = 0
            else

```

```

        {
            EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the
            failedIn(0);
            return(USBS_FAILED);
        }

case VERIFY_10:
    sensePtr = senseOk;
    checkForMedia();
    if (sensePtr == senseOk)
        return(ideReadCommand(1)); // ideReadCommand sets the OUT2BC = 0
    else
        {
            EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the
            failedIn(0);
            return(USBS_FAILED);
        }

case READ_CAPACITY:
    EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the CBW

    sensePtr = senseOk;
    checkForMedia();

    if (sensePtr == senseOk && waitForInBuffer() == USBS_PASSED)
        {
            IN2BUF[0] = ((BYTE *) &driveCapacity)[0];
            IN2BUF[1] = ((BYTE *) &driveCapacity)[1];
            IN2BUF[2] = ((BYTE *) &driveCapacity)[2];
            IN2BUF[3] = ((BYTE *) &driveCapacity)[3];
            IN2BUF[4] = (ATA_SECTOR_SIZE >> 24) & 0xff;
            IN2BUF[5] = (ATA_SECTOR_SIZE >> 16) & 0xff;
            IN2BUF[7] = (ATA_SECTOR_SIZE >> 0) & 0xff;
            IN2BUF[6] = (ATA_SECTOR_SIZE >> 8) & 0xff;
            IN2BC = sizeof(DWORD) * 2;
            status = USBS_PASSED;
        }
    else
        {
            failedIn(0);
            status = USBS_FAILED;
        }
    dataTransferLen -= sizeof(DWORD) * 2;
    return(status);

case TEST_UNIT_READY:
case PREVENT_ALLOW_MEDIUM_REMOVAL:
    EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the CBW
    if (dataTransferLen) // This command shouldn't have any data!
        failedIn(0);
    checkForMedia();
    if (sensePtr == senseOk)
        return(USBS_PASSED);
    else
        return(USBS_FAILED);

case REQUEST_SENSE:
    EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the CBW
    waitForInBuffer();
    mymemmovexx(IN2BUF, (char xdata *) sensePtr, sizeof(senseOk));
    IN2BC = min(sizeof(senseOk), dataTransferLen);
    dataTransferLen -= min(sizeof(senseOk), dataTransferLen);
    sensePtr = senseOk;
    return(USBS_PASSED);

case MODE_SELECT_06:
case MODE_SENSE_06:
    EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the CBW
    sensePtr = senseInvalidFieldInCDB;
    failedIn(0);
    return(USBS_FAILED);

case STOP_START_UNIT:
default:
    EPIO[OUT2BUF_ID].bytes = 0;           // relinquish control of the bulk buffer occupied by the CBW
    sensePtr = senseInvalidFieldInCDB;
    failedIn(0);

```

```

        return(USBS_FAILED);
    }
}

bit generalIDEOutCommand()
{
    BYTE cmd;

    cmd = OUT2BUF[0xf];

    switch (cmd)
    {
        case WRITE_10:
            sensePtr = senseOk;
            checkForMedia();
            if (sensePtr == senseOk)
                return(ideWriteCommand()); // ideWriteCommand sets the OUT2BC = 0
            else
            {
                EPIO[OUT2BUF_ID].bytes = 0; // relinquish control of the bulk buffer occupied by the
                return(USBS_FAILED);
            }

        default:
            EPIO[OUT2BUF_ID].bytes = 0; // relinquish control of the bulk buffer occupied by the CBW
            if (dataTransferLen)
                stalledP2OUT();
            return(USBS_FAILED);
            break;
    }
}

bit waitForInBuffer()
{
    while((EPIO[IN2BUF_ID].cntrl & bmEPBUSY)) // Wait for an available buffer from the host
        ;
    return(USBS_PASSED);
}

bit ideReadCommand(bit verify)
{
    BYTE driveStatus;
    WORD sectorcount;
    int timeout;
    BYTE i;
    DWORD dwLBA;

    writePIO8(ATA_DRIVESEL_REG, 0xe0);
    if (waitForBusyBit(PROCESS_CBW_TIMEOUT_RELOAD) == USBS_FAILED)
    {
        EPIO[OUT2BUF_ID].bytes = 0; // relinquish control of the bulk buffer occupied by the CBW
        return USBS_FAILED;
    }

    ((char *) &dwLBA)[0] = OUT2BUF[CBW_DATA_START+2];
    ((char *) &dwLBA)[1] = OUT2BUF[CBW_DATA_START+3];
    ((char *) &dwLBA)[2] = OUT2BUF[CBW_DATA_START+4];
    ((char *) &dwLBA)[3] = OUT2BUF[CBW_DATA_START+5];
    writePIO8(ATA_DRIVESEL_REG, 0xe0);
    writePIO8(ATA_LBA_LSB_REG, OUT2BUF[CBW_DATA_START+5]);
    writePIO8(ATA_LBA_2SB_REG, OUT2BUF[CBW_DATA_START+4]);
    writePIO8(ATA_LBA_MSB_REG, OUT2BUF[CBW_DATA_START+3]);
    writePIO8(ATA_DRIVESEL_REG, OUT2BUF[CBW_DATA_START+2] | 0xe0);
    // writePIO8(ATA_DRIVESEL_REG, 0xe0);
    // writePIO8(ATA_LBA_LSB_REG, ((char *) &dwLBA)[3]);
    // writePIO8(ATA_LBA_2SB_REG, ((char *) &dwLBA)[2]);
    // writePIO8(ATA_LBA_MSB_REG, ((char *) &dwLBA)[1]);
    // writePIO8(ATA_DRIVESEL_REG, ((char *) &dwLBA)[0] | 0xe0);

    EPIO[OUT2BUF_ID].bytes = 0; // relinquish control of the bulk buffer occupied by the CBW

    // This loop breaks up the 32 bit length into 8 bits * sectors.
    // For example, a read of 0x10000 turns into 0x80 sectors.
    while (dataTransferLen)
    {
        // Stuff the LBA registers
        // writePIO8(ATA_DRIVESEL_REG, 0xe0);

```

```

// writePIO8(ATA_LBA_LSB_REG, ((char *) &dwLBA)[3]);
// writePIO8(ATA_LBA_2SB_REG, ((char *) &dwLBA)[2]);
// writePIO8(ATA_LBA_MSB_REG, ((char *) &dwLBA)[1]);
// writePIO8(ATA_DRIVESEL_REG, ((char *) &dwLBA)[0] | 0xe0);

// First stuff the length register (number of sectors to read)
if (dataTransferLenMSW & 0xffff)
{
    writePIO8(ATA_SECTOR_COUNT_REG, 0); // 0 means 256 blocks of 512
    sectorcount = 0x100;
}
else
{
    sectorcount = dataTransferLenLSW/ATA_SECTOR_SIZE + (dataTransferLenMSW & 1) * 0x80;
    writePIO8(ATA_SECTOR_COUNT_REG, sectorcount); // divide len into blocks
}

dwLBA += sectorcount;

// Execute the read command
if (verify)
    writePIO8(ATA_COMMAND_REG, ATA_COMMAND_VERIFY_10);
else
    writePIO8(ATA_COMMAND_REG, ATA_COMMAND_READ_10);

// The verify command reads from the drive, but doesn't transfer data
// to us.
if (verify)
{
    if(waitForBusyBit(PROCESS_CBW_TIMEOUT_RELOAD) == USBS_FAILED)
        return(USBS_FAILED);
    else
        continue;
}

while (sectorcount--)
{
    timeout = PROCESS_CBW_TIMEOUT_RELOAD;

    do
    {
        driveStatus = readPIO8(ATAPI_STATUS_REG);
    }
    while(((driveStatus & (ATAPI_STATUS_BUSY_BIT | ATAPI_STATUS_DRQ_BIT)) != ATAPI_STATUS_DRQ_BIT) &&
!(timeout-- & 0x8000)); // DO-WHILE!!!

    if (!(timeout & 0x8000) && !(driveStatus & ATAPI_STATUS_ERROR_BIT))
    {
        // Normal case -- Got a sector. Send it to the host (in 8 chunks)
        for (i = 0; i < ATA_SECTOR_SIZE/MAX_BULK_PACKET_SIZE; i++)
        {
            while ((EPIO[IN2BUF_ID].cntrl & (bmEPBUSY | bmEPSTALL))) // Wait for an available buffer -
- DON'T timeout on USB i/f
            ;
            readPIO16toInBuf(); // always reads 0x40
            EPIO[IN2BUF_ID].bytes = MAX_BULK_PACKET_SIZE;
        }
        dataTransferLen -= ATA_SECTOR_SIZE; // Returned a full sector.
    }
    else
    {
        // Timeout case -- Assume the host has caught up, send a short packet, then goto the stall code.
        {
            readPIO8(ATAPI_ERROR_REG);
            IDEnop();
            failedIn(0);
            return(USBS_PASSED);
        }
    }
}
return(USBS_PASSED);
}

bit ideWriteCommand()
{
    BYTE savebc;
    BYTE driveStatus;
    WORD sectorcount;
    int timeout;
}

```

```

BYTE i;
DWORD dwLBA;

writePIO8(ATA_DRIVESEL_REG, 0xe0);
if (waitForBusyBit(PROCESS_CBW_TIMEOUT_RELOAD) == USBS_FAILED)
{
    EPIO[OUT2BUF_ID].bytes = 0;          // relinquish control of the bulk buffer occupied by the CBW
    return USBS_FAILED;
}

// Stuff the LBA registers
// writePIO8(ATA_LBA_LSB_REG, OUT2BUF[CBW_DATA_START+5]);
// writePIO8(ATA_LBA_2SB_REG, OUT2BUF[CBW_DATA_START+4]);
// writePIO8(ATA_LBA_MSB_REG, OUT2BUF[CBW_DATA_START+3]);
// writePIO8(ATA_DRIVESEL_REG, OUT2BUF[CBW_DATA_START+2] | 0xe0);
((char *) &dwLBA)[0] = OUT2BUF[CBW_DATA_START+2];
((char *) &dwLBA)[1] = OUT2BUF[CBW_DATA_START+3];
((char *) &dwLBA)[2] = OUT2BUF[CBW_DATA_START+4];
((char *) &dwLBA)[3] = OUT2BUF[CBW_DATA_START+5];
writePIO8(ATA_DRIVESEL_REG, 0xe0);
writePIO8(ATA_LBA_LSB_REG, ((char *) &dwLBA)[3]);
writePIO8(ATA_LBA_2SB_REG, ((char *) &dwLBA)[2]);
writePIO8(ATA_LBA_MSB_REG, ((char *) &dwLBA)[1]);
writePIO8(ATA_DRIVESEL_REG, ((char *) &dwLBA)[0] | 0xe0);

// We're done with the info in the CBW -- set it free to make room for the data packets
EPIO[OUT2BUF_ID].bytes = 0;          // relinquish control of the bulk buffer occupied by the CBW

// Send the command to the drive
// This loop breaks up the 32 bit length into 8 bits * sectors.
// For example, a read of 0x10000 turns into 0x80 sectors.
while (dataTransferLen)
{
    // Stuff the LBA registers
    writePIO8(ATA_DRIVESEL_REG, 0xe0);
    writePIO8(ATA_LBA_LSB_REG, ((char *) &dwLBA)[3]);
    writePIO8(ATA_LBA_2SB_REG, ((char *) &dwLBA)[2]);
    writePIO8(ATA_LBA_MSB_REG, ((char *) &dwLBA)[1]);
    writePIO8(ATA_DRIVESEL_REG, ((char *) &dwLBA)[0] | 0xe0);

    // First stuff the length register (number of sectors to write)
    if (dataTransferLenMSW & 0xffff)
    {
        writePIO8(ATA_SECTOR_COUNT_REG, 0x80);          // 0 means 256 blocks of 512, but 0x80 feels safer.
        sectorcount = 0x80;
    }
    else
    {
        sectorcount = dataTransferLenLSW/ATA_SECTOR_SIZE + (dataTransferLenMSW & 1) * 0x80;
        writePIO8(ATA_SECTOR_COUNT_REG, sectorcount);    // divide len into blocks
    }

    dwLBA += sectorcount;

    // Execute the write command
    writePIO8(ATA_COMMAND_REG, ATA_COMMAND_WRITE_10);

    while (sectorcount--)
    {
        timeout = PROCESS_CBW_TIMEOUT_RELOAD;

        do
        {
            driveStatus = readPIO8(ATAPI_STATUS_REG);
        }
        while(((driveStatus & (ATAPI_STATUS_BUSY_BIT | ATAPI_STATUS_DRQ_BIT)) != ATAPI_STATUS_DRQ_BIT) &&
!(timeout-- & 0x8000));    // DO-WHILE!!!

        if (!(timeout & 0x8000))
        {
            // Normal case -- Got a sector. Send it to the drive (in 8 chunks)
            for (i = 0; i < ATA_SECTOR_SIZE/MAX_BULK_PACKET_SIZE; i++)
            {
                while( (EPIO[OUT2BUF_ID].cntrl & bmEPBUSY))    // Wait for an available buffer from the
host
                    ;
                savebc = EPIO[OUT2BUF_ID].bytes;

                // Terminate xfer on receipt of short packet, otherwise drop

```

```

        // into streamlined case
        if (savebc < MAX_BULK_PACKET_SIZE)
        {
            writePIO16(ATAPI_DATA_REG, OUT2BUF, savebc);
            EPIO[OUT2BUF_ID].bytes = 0x40; // Give up the buffer
            dataTransferLen -= savebc + i * MAX_BULK_PACKET_SIZE;
            goto stopOnShortPacket;
        }
        else
        {
            writePIO16(ATAPI_DATA_REG, OUT2BUF, MAX_BULK_PACKET_SIZE);
            EPIO[OUT2BUF_ID].bytes = 0x40; // Give up the buffer
        }
        dataTransferLen -= ATA_SECTOR_SIZE; // Returned a full sector.
    }
    else
    {
        stalleP2OUT();
        return(USBS_FAILED);
    }
} // while (sectorcount)
} // While (dataTransferLen)

stopOnShortPacket:
    return(USBS_PASSED);
}

```

```

// Execute NOP on IDE device to clear error flag
void IDENop()
{
    BYTE count;
    BYTE driveStatus;

    // Pound the reset line
    hardwareReset();

    // Wait for the completion
    for (count = 0, driveStatus = readPIO8(ATAPI_STATUS_REG);
        count < 100 && (driveStatus & ATAPI_STATUS_BUSY_BIT);
        driveStatus = readPIO8(ATAPI_STATUS_REG))
    {
        EZUSB_Delay(50); // Wait 50 ms to be polite
    }

    writePIO8(ATA_DRIVESEL_REG, 0xe0);
    readPIO8(ATA_LBA_LSB_REG);
    readPIO8(ATA_LBA_2SB_REG);
    readPIO8(ATA_LBA_MSB_REG);
    readPIO8(ATA_DRIVESEL_REG);
}

```

```

;-----
; Copyright (c) 1999 Cypress Semiconductor, Inc. All rights reserved
;-----
; FASTXFER.SRC generated from: FASTXFER.C then modified by hand

```

\$NOMOD51

NAME FASTXFER

\$include (c:\anchor\ezusb\target\inc\ezregs.inc)

```

?PR?_writePIO8?FASTXFER          SEGMENT CODE
?XD?FASTXFER                     SEGMENT XDATA
?PR?_writePIO16?FASTXFER         SEGMENT CODE
?DT?_writePIO16?FASTXFER         SEGMENT DATA OVERLAYABLE
?PR?_readPIO8?FASTXFER           SEGMENT CODE
?PR?_readPIO16?FASTXFER          SEGMENT CODE
?DT?_readPIO16?FASTXFER          SEGMENT DATA OVERLAYABLE

```

```

PUBLIC  _readPIO16
PUBLIC  _readPIO8
PUBLIC  _writePIO16
PUBLIC  _writePIO8
PUBLIC  readPIO16toInBuf

```

```

RSEG ?XD?FASTXFER
retval?447: DS 4
scrap?242: DS 1

RSEG ?DT?_writePIO16?FASTXFER
?_writePIO16?BYTE:
inbuffer?344: DS 2

RSEG ?DT?_readPIO16?FASTXFER
?_readPIO16?BYTE:
inbuffer?549: DS 2

RESET_TOGGLE MACRO
ENDM

WRITE_ADDR MACRO
MOV A,R7 ; get the address from r7
ORL A,#02H ; OR in the control lines
MOV DPTR,#OUTC ; spit it out
MOVX @DPTR,A
ENDM

WRITE_ADDR_FOR_READ MACRO
MOV A,R7 ; get the address from r7
ORL A,#02H ; OR in the control lines
MOV DPTR,#OUTC ; spit it out
MOVX @DPTR,A
ENDM

RESTORE_ADDR_PINS MACRO
MOV DPL, #LOW(OUTC)
MOV A,#01fH
MOVX @DPTR,A
ENDM

STRETCH0 MACRO
MOV a, CKCON
ANL a, #0f8h
MOV CKCON, a
ENDM

STRETCH1 MACRO
MOV a, CKCON
ORL a, #1
MOV CKCON, a
ENDM

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; // Write the string to the given disk register or buffer
; void writePIO8(char addr, char indata)

RSEG ?PR?_writePIO8?FASTXFER
USING 0
_writePIO8:
;---- Variable 'addr?240' assigned to Register 'R7' ----
;---- Variable 'indata?241' assigned to Register 'R5' ----
;
; BYTE xdata scrap;
;
; scrap = indata;
MOV DPS, #0
MOV DPTR,#scrap?242
MOV A,R5
MOVX @DPTR,A
;
; // Write the address/chip selects
RESET_TOGGLE
WRITE_ADDR
;
; set stretch to 1
; STRETCH1
;
; AUTOPTR = &scrap;
MOV DPL, #LOW(AUTOPTRH)

```

```

MOV     A,#HIGH (scrap?242)
MOVX   @DPTR,A
INC    DPTR
MOV     A,#LOW (scrap?242)
MOVX   @DPTR,A
MOV     DPL, #LOW(AUTODATA)
movx   a, @dptr      ; looks like a read, but generates #FWR and drives data bus

;
;   set stretch to 0
;   STRETCH0

;   // Clear the address/chip selects
;   RESTORE_ADDR_PINS
;   RET
; END OF _writePIO8

;
; // Write the string to the given disk register or buffer
; void writePIO16(char addr, char xdata *inbuffer, BYTE count)

RSEG ?PR?_writePIO16?FASTXFER
USING 0
_writePIO16:
MOV     inbuffer?344,R4
MOV     inbuffer?344+01H,R5
;---- Variable 'addr?343' assigned to Register 'R7' ----
;---- Variable 'count?345' assigned to Register 'R3' ----
; {
;   // Write the address/chip selects
;   WRITE_ADDR
;
;   set stretch to 1
;   STRETCH1
;
;   AUTOPTR = inbuffer;
MOV     DPL, #LOW(AUTOPTRH)
MOV     A,inbuffer?344
MOVX   @DPTR,A
INC    DPTR
MOV     A,inbuffer?344+01H
MOVX   @DPTR,A
;   count += count & 1;           // round up to next word if it's an odd count
MOV     A,R3
ANL    A,#01H
ADD    A,R3
MOV     R3,A
;   count = 64-count;
CLR    C
MOV     A,#040H
SUBB   A,R3
;   #pragma asm
;   MOV     DPL, #LOW(AUTODATA)
;   MOV     DPL, #LOW(AUTODATA)
;
;   ;; perform programmed branch w/o using the dptr
;   add     a, #LOW(startOfWrite)
;   push   acc
;   mov    a, #0
;   addc  a, #HIGH(startOfWrite)
;   push  acc
;   ret
;   ;; Create 64 MOVX instructions to bring in the data
startOfWrite:
REPT   64
movx   a, @dptr      ; looks like a read, but generates #FWR and drives data bus
ENDM
;
;
;   set stretch to 0
;   STRETCH0

;   // Clear the address/chip selects
;   RESTORE_ADDR_PINS
;   RET
; END OF _writePIO16

```

```

;
; // Read a character from the given address.
; // Must read as words. Also must read one word to get started.
; // This means that we actually read 4x to get the one byte we care about.
; BYTE readPIO8(char addr)

        RSEG ?PR?_readPIO8?FASTXFER
        USING 0
_readPIO8:
;---- Variable 'addr?446' assigned to Register 'R7' ----
; {
;     xdata BYTE retval[4];
;
;     // Write the address/chip selects
MOV     DPS, #0    ;; Not the perfect place for this, but it preserves the optimization in fw and periph.
RESET_TOGGLE
WRITE_ADDR_FOR_READ
        STRETCH1
;
;
;     AUTOPTR = retval;
MOV     DPL, #LOW(AUTOPTRH)
MOV     A, #HIGH (inlbuf)
MOVX    @DPTR, A
INC     DPTR
MOV     A, #LOW (inlbuf)
MOVX    @DPTR, A
MOV     DPL, #LOW(AUTODATA)
movx    @dptr, a
;
; // Clear the address/chip selects
; STRETCH0
        RESTORE_ADDR_PINS
;
;     return(retval[0]);
MOV     DPTR, #inlbuf
MOVX    A, @DPTR
MOV     R7, A
;
?C0005:
        RET
; END OF _readPIO8

;
; // Read a string from the given disk register or buffer
; void readPIO16(char addr, char xdata *inbuffer, BYTE count)

        RSEG ?PR?_readPIO16?FASTXFER
        USING 0
readPIO16toInBuf:
;     WRITE_ADDR_FOR_READ
MOV     A, #(02h + 0)    ; OR in the control lines with the data register address (0) and CS value (2)
MOV     DPTR, #OUTC     ; spit it out
MOVX    @DPTR, A
;
MOV     DPL, #LOW(AUTOPTRH)
MOV     A, #HIGH(IN2BUF)    ; R4 is inbuffer MSB
MOVX    @DPTR, A
INC     DPTR
MOV     A, #LOW(IN2BUF)    ; R4 is inbuffer MSB
MOVX    @DPTR, A
;
MOV     DPL, #LOW(AUTODATA)
ljmp    startOfRead

_readPIO16:
;     MOV     inbuffer?549, R4
;     MOV     inbuffer?549+01H, R5
;---- Variable 'addr?548' assigned to Register 'R7' ----
;---- Variable 'count?550' assigned to Register 'R3' ----
; {
;     // Write the address/chip selects
;     RESET_TOGGLE
;     WRITE_ADDR_FOR_READ
;     STRETCH1
;
;
;     AUTOPTR = inbuffer;
MOV     DPL, #LOW(AUTOPTRH)
MOV     A, R4    ; R4 is inbuffer MSB

```

```

MOVX    @DPTR,A
INC     DPTR
MOV     A,R5 ; R5 is inbuffer LSB
MOVX    @DPTR,A

MOV     DPL, #LOW(AUTODATA)
;
;     count += count & 1;           // round up to next word if it's an odd count
MOV     A,R3
ANL     A,#01H
ADD     A,R3
MOV     R3, A
;     count = 62-(count-2);        // We get 2 for free at the end
MOV     A, #40H
CLR     C
SUBB    A,R3

; ; perform programmed branch w/o using the dptr
add     a, #LOW(startOfRead)
push   acc
mov     a, #HIGH(startOfRead)
addc   a, #0
push   acc
ret

; ; Create 62 MOVX instructions to bring in the data
startOfRead:
REPT    62
movx   @dptr, a
ENDM

;
; ; Final two reads
movx   @dptr, a
movx   @dptr, a

;
; // Clear the address/chip selects
; STRETCH0
; RESTORE_ADDR_PINS
; }
RET
; END OF _readPIO16

END

```

```

//-----
// Copyright (c) 1999 Cypress Semiconductor, Inc. All rights reserved
//-----
#pragma NOIV           // Do not generate interrupt vectors
//-----
// File:      periph.c
// Contents:  Hooks required to implement USB peripheral function.
//
// Copyright (c) 1997 AnchorChips, Inc. All rights reserved
//-----
// Switch compile from SCSI to IDE software -- TBD -- Can we determine it on the fly?
#define SHORT_PACKET_SENT_ON_FAILED_IN    0

#include <ezusb.h>
#include <ezregs.h>
#include <scsi.h>
#if FLASH
#include "flashcrd.h"
#else
  #if GPIF
    #include "gpif.h"
    #define AGGRESSIVELY_USE_TNG_FEATURES 1
    #include "tng.h"
  #else
    #include "fastxfer.h"
  #endif
#endif
#endif

#define MEMMOVE4(dest, src)      *((DWORD *) (dest)) = *((DWORD *) (src))
#define WRITE_ADDR_W(addr)     OUTC=addr|0xc0;

extern BOOL GotSUD;           // Received setup data flag

```

```

extern BOOL Sleep;

// States
#define UNCONFIGURED      0
#define WAIT_FOR_CBW     1
#define WAIT_TO_SEND_CSW 2
#define WAIT_FOR_OUT_DATA 3
#define WAIT_FOR_IN_DATA 4

xdata BYTE cbwTag[4];          // Tag from the most recent CBW packet
BYTE currentState;
xdata BYTE halfKBuffer[BUFFER_SIZE];
DWORD dataTransferLen;
DWORD driveCapacity;
bit scsi;
bit flash;

const BYTE code SCSIInquiryData[44] =
{
    0x00,          // = Device class
    //0x0e,        // Device class RBC
    0x00,          // = RMB bit is set by inquiry data
    0x00,          //
    0x01,          // = Data format = 1
    0x00,          // = Additional length (changed to 0 from 0x75)
    0x00, 0x00, 0x00, //
    0x53, 0x61, 0x6e, 0x44, 0x69, 0x73, 0x6b, 0x20, // = Manufacturer
    0x49, 0x6d, 0x61, 0x67, 0x65, 0x4d, 0x61, 0x74, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, // = Product (Zip
    100)
    0x30, 0x31, 0x2e, 0x30, // = Revision
    0x30, 0x39, 0x2f, 0x32, 0x34, 0x2f, 0x39, 0x38, // = Vendor unique (chopped off)
};

//-----
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//-----

void TD_Init(void)          // Called once at startup
{
    currentState = UNCONFIGURED;
    initPorts();           // Different for GPIF and PIO

    // Enable endpoint 2 in, and endpoint 2 out
    IN07VAL = bmEP2;       // Validate all EP's
    OUT07VAL = bmEP2;

    // Enable double buffering on endpoint 2 in, and endpoint 2 out
    USBPAIR = 0x09;

    // stretch = 0
    CKCON &= ~7;

    // EZUSB_Delay(8000);

    resetATAPIDevice();
    if (SCSITestUnitReady())
        SCSITestUnitReady();

    if (!flash)
    {
        ATAPIIdDevice();    // Get serial number
        if (SCSITestUnitReady())
            SCSITestUnitReady();

        if (scsi)
            SCSIInquiryToATAPI(); // identify CD ROM device vs Zip
        else
        {
            WORD convert;
            convert = &IntrfcSubClass;
            // *((BYTE xdata *) convert) = USB_MS_RBC_SUBCLASS;
        }
    }
}

char const code usbcString[] = "USBC";
void TD_Poll(void)          // Called repeatedly while the device is idle

```

```

{
    BYTE    count;

    if( !(EPIO[OUT2BUF_ID].cntrl & bmEPBUSY) )        // Is there something in the OUT2BUF buffer,
    {
        // Check for "USBC"
        // if (memcmp(OUT2BUF, "USBC",4))
        if ( *((DWORD xdata *)OUT2BUF) != *((DWORD xdata *) usbcString))
        {
            // error -- Stall the endpoint
            stalleP2OUT();
        }
        else
        {
            count = EPIO[OUT2BUF_ID].bytes;
            if (count < OUT2BUF[CBW_CBW_LEN]+CBW_DATA_START)
                // Error -- Stall the endpoint
                stalleP2OUT();
            else
                // Good packet, forward to the device.
                processCBW();
        }
    }
}

#if 0
BOOL TD_Suspend(void)        // Called before the device goes into suspend mode
{
    return(TRUE);
}

BOOL TD_Resume(void)        // Called after the device resumes
{
    return(TRUE);
}
#endif

//-----
// Device Request hooks
// The following hooks are called by the end point 0 device request parser.
//-----

#if 0
BOOL DR_GetDescriptor(void)
{
    return(TRUE);
}

BOOL DR_SetConfiguration(void)    // Called when a Set Configuration command is received
{
    Configuration = SETUPDAT[2];
    return(TRUE);                // Handled by user code
}

BOOL DR_GetConfiguration(void)    // Called when a Get Configuration command is received
{
    IN0BUF[0] = Configuration;
    EZUSB_SET_EP_BYTES(IN0BUF_ID,1);
    return(TRUE);                // Handled by user code
}

BOOL DR_SetInterface(void)        // Called when a Set Interface command is received
{
    AlternateSetting = SETUPDAT[2];
    return(TRUE);                // Handled by user code
}

BOOL DR_GetInterface(void)        // Called when a Set Interface command is received
{
    IN0BUF[0] = AlternateSetting;
    EZUSB_SET_EP_BYTES(IN0BUF_ID,1);
    return(TRUE);                // Handled by user code
}

BOOL DR_GetStatus(void)
{
    return(TRUE);
}

```

```

BOOL DR_ClearFeature(void)
{
    return(TRUE);
}

BOOL DR_SetFeature(void)
{
    return(TRUE);
}

BOOL DR_VendorCmnd(void)
{
    return(TRUE);
}
#endif

//-----
// USB Interrupt Handlers
// The following functions are called by the USB interrupt jump table.
//-----

// Setup Data Available Interrupt Handler
void ISR_Sudav(void) interrupt 0
{
    GotSUD = TRUE;           // Set flag
    SetupCommand();
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmsUDAV;       // Clear SUDAV IRQ
}

// Not used
// void ISR_Sof(void) interrupt 0
//{
//    EZUSB_IRQ_CLEAR();
//    USBIRQ = bmSOF;        // Clear SOF IRQ
//}

void ISR_Ures(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
    USBPAIR = 0x00;         // Resets toggle?

    EPIO[OUT2BUF_ID].bytes = 0; // Make sure there is no data waiting for us
    EPIO[OUT3BUF_ID].bytes = 0; // clear the double buffer too.
    // clear the stall and busy bits that may be set
    EPIO[OUT2BUF_ID].cntrl = 0;
    EPIO[IN2BUF_ID].cntrl = bmEPBUSY; // (write 1 to clear)
    EPIO[IN3BUF_ID].cntrl = bmEPBUSY;
    if((EPIO[IN2BUF_ID].cntrl & bmEPBUSY)) // BUGBUG -- Sanity check
        while (1)
        {
        }
    USBPAIR = 0x09;

    USBIRQ = bmURES;        // Clear URES IRQ

    if (currentState != UNCONFIGURED)
    {
        // force a soft reset after the ired.
        softReset();
    }
}

void ISR_Spare(void) interrupt 0
{
}

void ISR_Susp(void) interrupt 0
{
    Sleep = TRUE;
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUSP;
}

// void ISR_Ep2in(void) interrupt 0
// {
// }

```

```

// void ISR_Ep2out(void) interrupt 0
// {
// }

void processCBW()
{
    // Save the tag for use in the response
    mymemmovexx(cbwTag, OUT2BUF+CBW_TAG, 4);

    // Get the length (convert from little endian)
    *(((BYTE *) &dataTransferLen)+0) = (OUT2BUF+CBW_DATA_TRANSFER_LEN_LSB)[3]; // "Residue"
    *(((BYTE *) &dataTransferLen)+1) = (OUT2BUF+CBW_DATA_TRANSFER_LEN_LSB)[2]; // "Residue"
    *(((BYTE *) &dataTransferLen)+2) = (OUT2BUF+CBW_DATA_TRANSFER_LEN_LSB)[1]; // "Residue"
    *(((BYTE *) &dataTransferLen)+3) = (OUT2BUF+CBW_DATA_TRANSFER_LEN_LSB)[0]; // "Residue"

    if (OUT2BUF[CBW_FLAGS] & CBW_FLAGS_DIR_BIT || !dataTransferLen)
    {
        if (scsi)
            sendUSBS(generalSCSIInCommand());
        else
            sendUSBS(generalIDEInCommand());
    }
    else
    {
        if (scsi)
            sendUSBS(generalSCSIOutCommand());
        else
            sendUSBS(generalIDEOutCommand());
    }
    currentState = WAIT_FOR_CBW;
}

// Stalls EP2OUT endpoint.
void stalledEP2OUT()
{
    EZUSB_STALL_EP(OUT2BUF_ID);
}

// Read the Inquiry info into our internal data structures.
// NOT prompted by the host.
#define INQUIRY_LEN 0x2c
const char code inquiryCommand[12] = { 0x12, 0x00, 0x00, 0x00, INQUIRY_LEN, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
void SCSIInquiryToATAPI()
{
    int i;
    bit result;
    result = sendSCSICommand(0, (char xdata *) inquiryCommand);
    if (result == USBS_FAILED)
    {
        // failedIn(0); This is an internal command, just leave if it fails.
        return;
    }

    result = waitforIntrq();
    readPIO16toXdata(ATAPI_DATA_REG, halfKBuffer, INQUIRY_LEN);

    // for (i = 0; i < SCSI_INQUIRY_SERIAL_LEN; i++)
    // ((BYTE xdata *)serialNumber)[i*2] = halfKBuffer[i+SCSI_INQUIRY_SERIAL];

    i = &IntrfcSubClass;

    if (halfKBuffer[SCSI_INQUIRY_DEVICE_CLASS] == 5)
        *((char xdata *) i) = USB_MS_CD_ROM_SUBCLASS;
    else
        *((char xdata *) i) = USB_MS_SCSI_TRANSPARENT_SUBCLASS;

    readPIO8(ATAPI_ERROR_REG);
}

#define SENSE_LEN 18
const char code testUnitReady[12] = { 0x00, 0x00, 0x00, 0x00, 00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
const char code requestSense[12] = { 0x03, 0x00, 0x00, 0x00, SENSE_LEN, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
// Send a TestUnitReady command to the device.
// If the device fails the testUnitReady command, return the sense code, else return 0
BYTE SCBITestUnitReady()

```

```

{
    bit result;

    if (!scsi)
        return 0;

    result = sendSCSICommand(0, (char xdata *) testUnitReady);
    if (result == USB_FAILED)
    {
        result = sendSCSICommand(0, (char xdata *) requestSense);
        if (result == USB_FAILED)
            while (1)
            {
            }

        result = waitForIntrq();

        readPIO16toXdata(ATAPI_DATA_REG, halfKBuffer, SENSE_LEN);
        return(halfKBuffer[12]);
    }

    return(0);
}

void sendUSBS(BYTE passOrFail)
{
    WORD timeout = PROCESS_CBW_TIMEOUT_RELOAD;
    bit done = 0;

    // While there is more data left to process, get more buffers and process them
    // Perform IN operation
    while (--timeout && !done)
    {
        if (!(EPIO[IN2BUF_ID].cntrl & (bmEPBUSY | bmEPSTALL))) // Wait for an available buffer
        {
            // Fill the buffer & send the data back to the host
            IN2BUF[0] = 'U'; // Believe it or not, this is pretty efficient!
            IN2BUF[1] = 'S';
            IN2BUF[2] = 'B';
            IN2BUF[3] = 'S';
            mymemmovexx(IN2BUF+4, cbwTag, 4);
            // have to store LSB first
            IN2BUF[8+0] = ((BYTE *)&dataTransferLen)[3]; // "Residue"
            IN2BUF[8+1] = ((BYTE *)&dataTransferLen)[2]; // "Residue"
            IN2BUF[8+2] = ((BYTE *)&dataTransferLen)[1]; // "Residue"
            IN2BUF[8+3] = ((BYTE *)&dataTransferLen)[0]; // "Residue"

            *((BYTE xdata *) (IN2BUF+12)) = passOrFail; // Status
            EPIO[IN2BUF_ID].bytes = 13;
            done = 1;
        }
    }

    if (!timeout)
    {
        // BUGBUG -- I want to notice this event
        EZUSB_STALL_EP(IN2BUF_ID);
        while (1)
        {
        }
        return;
    }
}

void failedIn(bit shortPacketSent)
{
    #if 0
    // If the host is expecting data, terminate with a short packet.
    if (dataTransferLen)
    {
        The iomega drive doesn't do this, let's see if we can get away with it.
        if (!shortPacketSent)
        {
            while(EPIO[IN2BUF_ID].cntrl & bmEPBUSY) // Wait for an available buffer
                ;
            EPIO[IN2BUF_ID].bytes = 0;
        }
    }
    #endif
}

```

```

        while(EPIO[IN2BUF_ID].cntrl & bmEPBUSY)    // Wait for the message to go out
            ;
    }
}
#endif

// Stall if the host is still expecting data
if (dataTransferLen)
    EZUSB_STALL_EP(IN2BUF_ID);
}

// Read data from the drive
// Issues repeated calls to readPIO16 to pull in multiple blocks
// of data from the drive
// Returns amount of data NOT read.
WORD readPIO16toXdata(char addr, char xdata *inbuffer, WORD count)
{
    WORD driveDataLen = 0;
    BYTE driveStatus;
    WORD readLen;
    WORD timeout = PROCESS_CBW_TIMEOUT_RELOAD;

    {
        for (driveStatus = 0; !(driveStatus & ATAPI_STATUS_DRQ_BIT) && timeout-- > 0; )
            driveStatus = readPIO8(ATAPI_STATUS_REG);
        if (!timeout)
            return(count);

        if (scsi)
        {
            driveDataLen = getDriveDataLen();
            count = min(count, driveDataLen);
        }
        else
        {
            driveDataLen = ATA_SECTOR_SIZE;
            count = min(count, driveDataLen);
            driveDataLen -= count;
        }
    }

    while (count)
    {
        readLen = min(count, 0x40);    // Read and write routines are limited to one USB buffer size
        readPIO16(addr, inbuffer, readLen);
        count -= readLen;
        inbuffer += readLen;
    }

    // For IDE, must empty the buffer after the relevant data has been read.
    if (!scsi)
        while (driveDataLen)
        {
            char xdata scrap[2];
            readPIO16(addr, scrap, 2);
            driveDataLen -= 2;
        }
    return(count);
}

bit waitForBusyBit(WORD timeout)
{
    BYTE driveStatus;

    do
    {
        driveStatus = readPIO8(ATAPI_STATUS_REG);
    }
    while((driveStatus & (ATAPI_STATUS_BUSY_BIT)) && --timeout);    // Do-while

    if (!timeout || (driveStatus & ATAPI_STATUS_ERROR_BIT))
        return(USBS_FAILED);
    else
        return(USBS_PASSED);
}

```

```

void mymemmovexx(BYTE xdata * dest, BYTE xdata * src, BYTE len)
{
    while (len--)
    {
        *dest++ = *src++;
    }
}

```

```

//-----
// File:      fw.c
// Contents:  Firmware frameworks task dispatcher and device request parser
//           source.
//
// Copyright (c) 1999 Cypress Semiconductor, Inc. All rights reserved
//-----
#include <REG320.H>
#include <string.H>
#include "ezusb.h"
#include "ezregs.h"
#if GPIF
#define AGGRESSIVELY_USE_TNG_FEATURES 1
#include "tng.h"
#include "gpif.h"
#endif
#include "atapi.h"
//-----
// Random Macros
//-----
#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)>(b))?(a):(b))

//-----
// Constants
//-----
#define DELAY_COUNT          0x9248L*5      // Delay for 1 sec

// USB constants
// Class specific setup commands
#define SC_BOMS_RESET        (0x21)        // Hard/soft depends on wValue field 0 = hard

#define BACKGROUD_SETUPS 0

//-----
// Global Variables
//-----
volatile BOOL  GotSUD  ;
BOOL          Rwen    ;
BOOL          Selfpwr ;
volatile BOOL  Sleep  ;                      // Sleep mode enable flag
BYTE  Configuration; // Current configuration
BYTE  AlternateSetting; // Alternate settings
//-----
// Prototypes
//-----
void SetupCommand(void);
void TD_Init(void);
void TD_Poll(void);
#define TD_Suspend()  1
#define TD_Resume()
BYTE epid(BYTE inEP);

#define DR_GetDescriptor()  1
BOOL DR_SetConfiguration(void);
BOOL DR_GetConfiguration(void);
BOOL DR_SetInterface(void);
BOOL DR_GetInterface(void);
#define DR_GetStatus()  1
#define DR_ClearFeature()  1
#define DR_SetFeature()  1
#define DR_VendorCmnd()  1

xdata char comparePattern[8];
const char code originalPattern[8] = "Cypress";

//-----
// Code

```

```

//-----
void main(void)
{
    DWORD j;
    BYTE i;

    // Initialize Global States
    Sleep = FALSE;           // Disable sleep mode
    Rwuen = FALSE;          // Disable remote wakeup
    Selfpwr = TRUE;         // Disable self powered
    GotSUD = FALSE;         // Clear "Got setup data" flag

    for (i = 0; i < 8; i++)
    {
        if (comparePattern[i] != ((char xdata *) originalPattern)[i])
        {
            #if GPIF
            OUTA &= ~GPIF_DISCON_;           // Disconnect via port pin
            #else
            USBCS &= ~bmDISCOE;             // Tristate the disconnect pin
            #endif
            USBCS |=bmDISCON;               // Set this high to disconnect USB
            mymemmovexx(comparePattern, (char xdata *)originalPattern, 8); // prevent disconnect on restart
            break;
        }
    }
    // Initialize USB Core to unconfigured state
    // DisableEPs();

    // Initialize user device -- This will wait until the drive is detected
    // There is at least 1 second of delay in here, so the host should notice us being gone
    TD_Init();

    USBIRQ = 0xff;           // Clear any pending USB interrupt requests. They're for our old life.
    // IN07IRQ = 0xff;       // clear old activity
    // OUT07IRQ = 0xff;
    EZUSB_IRQ_CLEAR();

    EZUSB_IRQ_ENABLE();     // Enable USB interrupt (INT2)
    EZUSB_ENABLE_RSMIRQ();  // Wake-up interrupt

    USBIEN = bmSUDAV | bmSUSP | bmURES; // Enable SUDAV interrupt
    EA = 1;                  // Enable 8051 interrupts

    USBCS |=bmRENUM;        // set renumerate bit so the 8051 gets the messages
    USBCS &= ~bmDISCON;     // reconnect USB
    USBCS |=bmDISCOE;       // Release Tristated disconnect pin
    #if GPIF
    OUTA |= GPIF_DISCON_;
    #endif

    // NOTE: The device will continue to renumerate until it receives a setup
    // packet. This fixes a microsoft USB bug that loses connect events during
    // initial USB device driver configuration dialog box.
    while(!GotSUD)
    {
        if(!GotSUD)
            EZUSB_Discon(TRUE); // renumerate until setup received
        for(j=0; (j<DELAY_COUNT) && (!GotSUD); ++j)
            ;
    }

    // Task Dispatcher
    while(TRUE) // Main Loop
    {
        #if BACKGROUND_SETUPS
        if(GotSUD) // Wait for SUDAV
        {
            SetupCommand(); // Implement setup command
            GotSUD = FALSE; // Clear SUDAV flag
        }
        #endif
        // Poll User Device
        if(Sleep) // The sleep flag is set by the Suspend ISR
        {
            // NOTE: Idle mode stops the processor clock. There are only two
            // ways out of idle mode, the WAKEUP pin, and detection of the USB
            // resume state on the USB bus. The timers will stop and the
            // processor will not wake up on any interrupts.

```

```

    if(TD_Suspend())
    {
        // need to check remote wake-up enable somewhere
        Sleep = FALSE;        // Re-arm the suspend ISR before we go to sleep -- Prevents race condition.
        EZUSB_Susp();        // Place processor in idle mode. This stops the clocks

        EZUSB_Resume();
        TD_Resume();
    }
}

TD_Poll();
}

// Decode and implement Setup Command
// Called from the setup ISR now!
void SetupCommand(void)
{
    BYTE    setupCopy[8];
    bit     stalleP0 = 0;
    void    xdata *dscr_ptr;
    {
        register BYTE i;

        for (i = 0; i < 8; i++)
        {
            setupCopy[i] = SETUPDAT[i];
        }
    }

    switch(setupCopy[1])
    {
        case SC_GET_DESCRIPTOR:        // *** Get Descriptor
            if(DR_GetDescriptor())
                switch(setupCopy[3])
                {
                    case GD_DEVICE:                // Device
                        SUDPTRH = MSB(&DeviceDscr);
                        SUDPTL = LSB(&DeviceDscr);
                        break;
                    case GD_CONFIGURATION:        // Configuration
                        if(dscr_ptr = (void xdata *)EZUSB_GetConfigDscr(setupCopy[2]))
                        {
                            SUDPTRH = MSB(dscr_ptr);
                            SUDPTL = LSB(dscr_ptr);
                        }
                        else
                            stalleP0 = 1;        // Stall End Point 0
                        break;
                    case GD_STRING:                // String
                        if(dscr_ptr = (void *)EZUSB_GetStringDscr(setupCopy[2]))
                        {
                            // Workaround for rev D bug
                            STRINGDSCR xdata *sdp;
                            BYTE len;

                            sdp = (STRINGDSCR xdata *) dscr_ptr;

                            len = sdp->length;

                            if (len > setupCopy[6])
                                len = setupCopy[6];        //limit to the requested length

                            while (len)
                            {
                                BYTE i;
                                for (i = 0; i < min(len, 64); i++)
                                {
                                    INOBUF[i] = ((BYTE xdata *)sdp)[i];
                                }
                                EZUSB_SET_EP_BYTES(INOBUF_ID,min(len,64));        //set length and arm Endpoint
                                len -= min(len,64);

                                // Wait for it to go out (Rev C and above)
                                while(EPOCS & 0x04)
                                    ;
                            }
                        }
                        // Clear the HS-nak bit

```

```

        EPOCS = bmHS;
    }
    else
        stalleP0 = 1;    // Stall End Point 0
        break;
    default:    // Invalid request
        stalleP0 = 1;    // Stall End Point 0
    }
    break;
case SC_GET_INTERFACE:    // *** Get Interface
    INOBUF[0] = AlternateSetting;
    EZUSB_SET_EP_BYTES(INOBUF_ID,1);
    break;
case SC_SET_INTERFACE:    // *** Set Interface
    AlternateSetting = setupCopy[2];
    break;
case SC_SET_CONFIGURATION:// *** Set Configuration
    Configuration = setupCopy[2];
    break;
case SC_GET_CONFIGURATION:// *** Get Configuration
    INOBUF[0] = Configuration;
    EZUSB_SET_EP_BYTES(INOBUF_ID,1);
    break;
case SC_GET_STATUS:    // *** Get Status
    if(DR_GetStatus())
        switch(setupCopy[0])
        {
            case GS_DEVICE:    // Device
                INOBUF[0] = ((BYTE)Rwuen << 1) | (BYTE)Selfpwr;
                INOBUF[1] = 0;
                EZUSB_SET_EP_BYTES(INOBUF_ID,2);
                break;
            case GS_INTERFACE:    // Interface
                INOBUF[0] = 0;
                INOBUF[1] = 0;
                EZUSB_SET_EP_BYTES(INOBUF_ID,2);
                break;
            case GS_ENDPOINT:    // End Point
                INOBUF[0] = EPIO[epid(setupCopy[4])].cntrl & ~bmEPBUSY;
                INOBUF[1] = 0;
                EZUSB_SET_EP_BYTES(INOBUF_ID,2);
                break;
            default:    // Invalid Command
                stalleP0 = 1;    // Stall End Point 0
        }
    break;
case SC_CLEAR_FEATURE:    // *** Clear Feature
    if(DR_ClearFeature())
        switch(setupCopy[0])
        {
            case FT_DEVICE:    // Device
                if(setupCopy[2] == 1)
                    Rwuen = FALSE;    // Disable Remote Wakeup
                else
                    stalleP0 = 1;    // Stall End Point 0
                break;
            case FT_ENDPOINT:    // End Point
                if(setupCopy[2] == 0)
                {
                    EZUSB_RESET_DATA_TOGGLE(setupCopy[4]);
                    if (epid(setupCopy[4]) == IN2BUF_ID || epid(setupCopy[4]) == OUT1BUF_ID)    //
                    Apple MSD version 1.3.5 mistakenly clears stalls on OUT1, not the EP reported to it.
                    {
                        USBPAIR = 0x00;    // Resets toggle?
                        EPIO[IN3BUF_ID].cntrl = bmEPBUSY;    // Clear any pending data (write 1 to clear)
                        EPIO[IN2BUF_ID].cntrl = bmEPBUSY;    // This also clears the stall bit(s)
                        USBPAIR = 0x09;
                    }
                }
                else if (epid(setupCopy[4]) == OUT2BUF_ID)
                {
                    USBPAIR = 0x00;    // Resets toggle?
                    EPIO[OUT3BUF_ID].cntrl = bmEPBUSY;    // Clear any pending data (write 1 to clear)
                    EPIO[OUT2BUF_ID].cntrl = bmEPBUSY;    // This also clears the stall bit(s)
                    USBPAIR = 0x09;
                }
            }
            else
                EZUSB_UNSTALL_EP( epid(setupCopy[4]) );
        }
    else

```

```

        stalleP0 = 1;        // Stall End Point 0
        break;
    }
    break;
case SC_SET_FEATURE:        // *** Set Feature
    if(DR_SetFeature())
        switch(setupCopy[0])
        {
            case FT_DEVICE:                // Device
                if(setupCopy[2] == 1)
                    Rwuen = TRUE;        // Enable Remote Wakeup
                else
                    stalleP0 = 1;        // Stall End Point 0
                    break;
            case FT_ENDPOINT:                // End Point
                if(setupCopy[2] == 0)
                    EZUSB_STALL_EP( epid(setupCopy[4]) );
                else
                    stalleP0 = 1;        // Stall End Point 0
                    break;
        }
    break;
default:                    // *** Invalid Command
    if(DR_VendorCmnd())
        stalleP0 = 1;        // Stall End Point 0
}

if (stalleP0)
    EZUSB_STALL_EP0();        // Stall End Point 0

// Acknowledge handshake phase of device request
// Required for rev C does not effect rev B
EPOCS |= bmBIT1;
}

// Wake-up interrupt handler
void resume_isr(void) interrupt WKUP_VECT
{
    EZUSB_CLEAR_RSMIRQ();
}

BYTE epid(BYTE inEP)
{
    return(EPID(inEP));
}

```

```

;;-----
;; File:      dscr.a51
;; Contents:  This file contains descriptor data tables.
;;
;; Copyright (c) 1999 Cypress Semiconductor, Inc. All rights reserved
;;-----

```

```

DSCR_DEVICE equ 1    ;; Descriptor type: Device
DSCR_CONFIG equ 2    ;; Descriptor type: Configuration
DSCR_STRING equ 3    ;; Descriptor type: String
DSCR_INTRFC equ 4    ;; Descriptor type: Interface
DSCR_ENDPNT equ 5    ;; Descriptor type: Endpoint

```

```

ET_CONTROL equ 0     ;; Endpoint type: Control
ET_ISO      equ 1     ;; Endpoint type: Isochronous
ET_BULK     equ 2     ;; Endpoint type: Bulk
ET_INT      equ 3     ;; Endpoint type: Interrupt

```

```

public      DeviceDscr, ConfigDscr, StringDscr, UserDscr, SerialNumber
public      IntrfcSubClass

```

```

DSCR      SEGMENT CODE

```

```

;;-----
;; Global Variables
;;-----
;; Note: This segment must be located in on-part memory.
    rseg DSCR        ;; locate the descriptor table anywhere below 8K
DeviceDscr: db deviceDscrEnd-DeviceDscr    ;; Descriptor length
            db DSCR_DEVICE;                ;; Descriptor type
            dw 1001H                        ;; Specification Version (BCD)
            db 00H                            ;; Device class

```

```

db 00H    ;; Device sub-class
db 00H    ;; Device sub-sub-class
db 64     ;; Maximum packet size
dw 4705H  ;; Vendor ID
dw 1028H  ;; Product ID - set to new id = 2810
dw 0100H  ;; Product version ID
db 1      ;; Manufacturer string index
db 3      ;; Product string index
db 2      ;; Serial number string index
db 1      ;; Number of configurations
deviceDscrEnd:

ConfigDscr:db ConfigDscrEnd-ConfigDscr    ;; Descriptor length
db DSCR_CONFIG ;; Descriptor type
db 32      ;; Configuration + End Points length (LSB)
db 00      ;; Configuration length (MSB)
db 1      ;; Number of interfaces
db 1      ;; Interface number
db 0      ;; Configuration string
db 01000000b ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
db 0      ;; Power requirement (div 2 ma)
ConfigDscrEnd:

IntrfcDscr:
db IntrfcDscrEnd-IntrfcDscr    ;; Descriptor length
db DSCR_INTRFC ;; Descriptor type
db 0      ;; Zero-based index of this interface
db 0      ;; Alternate setting
db 2      ;; Number of end points
db 08H    ;; Interface class
IntrfcSubClass:
db 06H    ;; Interface sub class -- 02 = CD-ROM, 04 = Floppy (UFI), 05 = Floppy (SFF8070I), 06 = SCSI
transparent
db 50H    ;; Interface sub sub class -- Bulk only (from zip drive)
db 0      ;; Interface descriptor string index
IntrfcDscrEnd:

EpInDscr:
db EpInDscrEnd-EpInDscr    ;; Descriptor length
db DSCR_ENDPNT ;; Descriptor type
db 82H    ;; Endpoint number, and direction
db ET_BULK ;; Endpoint type
db 40H    ;; Maximum packet size (LSB)
db 00H    ;; Max packet size (MSB)
db 00H    ;; Polling interval
EpInDscrEnd:

EpOutDscr:
db EpOutDscrEnd-EpOutDscr    ;; Descriptor length
db DSCR_ENDPNT ;; Descriptor type
db 02H    ;; Endpoint number, and direction
db ET_BULK ;; Endpoint type
db 40H    ;; Maximum packet size (LSB)
db 00H    ;; Max packet size (MSB)
db 00H    ;; Polling interval
EpOutDscrEnd:

StringDscr:
StringDscr0:
db StringDscr0End-StringDscr0    ;; String descriptor length
db DSCR_STRING
db 09H,04H
StringDscr0End:

StringDscr1:
db StringDscr1End-StringDscr1    ;; String descriptor length
db DSCR_STRING
db 'A',00
db 'n',00
db 'c',00
db 'h',00
db 'o',00
db 'r',00
db ' ',00
db 'C',00
db 'h',00
db 'i',00
db 'p',00

```

```

        db 's',00
StringDscr1End:

StringDscr2:
        db StringDscr2End-StringDscr2    ;; String descriptor length
        db DSCR_STRING
SerialNumber:
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '0',00
        db '1',00
StringDscr2End:

StringDscr3:
        db StringDscr3End-StringDscr3    ;; String descriptor length
        db DSCR_STRING
        db 'A',00
        db 'n',00
        db 'c',00
        db 'h',00
        db 'o',00
        db 'r',00
        db ' ',00
        db 'A',00
        db 'T',00
        db 'A',00
        db 'P',00
        db 'I',00
        db ' ',00
        db 'B',00
        db 'r',00
        db 'i',00
        db 'd',00
        db 'g',00
        db 'e',00
StringDscr3End:

UserDscr:
        dw 0000H
        end

```

```

NAME          USBJumpTbl

extrn         code (ISR_Sudav, ISR_Susp, ISR_Ures, ISR_Spare)
public       USB_AutoVector, USB_Jump_Table
;-----
; Interrupt Vectors
;-----
        CSEG      AT 43H
USB_AutoVector equ $ + 2
        ljmp     USB_Jump_Table ; Autovector will replace byte 45

;-----
; USB Jump Table
;-----
?PR?USB_JUMP_TABLE?USBJT segment code page ; Place jump table on a page boundary
        RSEG     ?PR?USB_JUMP_TABLE?USBJT ; autovector jump table
USB_Jump_Table:
        ljmp     ISR_Sudav ; Setup Data Available
        db      0
        ljmp     ISR_Spare ; Start of Frame
        db      0
        ljmp     ISR_Spare ; Setup Data Loading -- not used
        db      0

```

```
    ljmp  ISR_Susp   ; Global Suspend
    db 0
    ljmp  ISR_Ures   ; USB Reset
;
;
;    ljmp  ISR_Spare ; Reserved
;    db 0
;
;    ljmp  ISR_Spare ; End Point 0 In
;    db 0
;
;    ljmp  ISR_Spare ; End Point 0 Out
;    db 0
;
;    ljmp  ISR_Spare ; End Point 1 In
;    db 0
;
;    ljmp  ISR_Spare ; End Point 1 Out
;    db 0
;
;    ljmp  ISR_Spare
;    db 0
;
;    ljmp  ISR_Spare
;    db 0
;
;
end
```