

# **USB Development Board 2**

**Kyumsung Lee  
Nikhil Jayakumar**

**Univ of Colorado at Boulder.**

## Table of Contents:

<b><u>DEVELOPMENT BOARD 2 - INTRODUCTION</u></b> .....	<b>3</b>
<b><u>LIST OF SWITCHES, JUMPERS, TESTING POINTS, CONNECTORS USED</u></b> .....	<b>3</b>
<u>Pushbuttons</u> .....	3
<u>Jumpers</u> .....	3
<u>Testing points</u> .....	4
<u>Connectors</u> .....	5
<b><u>SCHEMATICS</u></b> .....	<b>7</b>
<b><u>PCB LAYOUT</u></b> .....	<b>12</b>
<u>PCB Layout – All Layers:</u> .....	12
<u>PCB Layout - Top Layer</u> .....	13
<u>PCB Layout - Bottom Layer</u> .....	14
<u>PCB Layout – Solder Mask Top</u> .....	15
<u>PCB Layout – Solder Mask Bottom</u> .....	16
<u>PCB Layout – Silk Screen Top</u> .....	17
<u>PCB Layout - Drill Drawing</u> .....	19
<b><u>DEVELOPING A HID</u></b> .....	<b>20</b>
<u>HID Report Descriptor:</u> .....	20
<u>The Hardware:</u> .....	20
<u>The complete (frameworks) code for USB Human Interface Device:</u> .....	21
<b><u>MISC. CODE</u></b> .....	<b>43</b>
<u>Code to Debug Via serial port:</u> .....	43
<u>Compact Flash test:</u> .....	44
<u>8-bit_test.c: This file tests 8-bit GPIF read and writes to CompactFlash.</u> .....	44
<u>16bittest.c: This code tests 16-bit GPIF Reads and Writes to CompactFlash</u> .....	60
<u>fifoest.c: This code tests 16-bit FIFO Reads and Writes</u> .....	74
<u>The complete (frameworks) code for USB Compact Flash Mass Storage Device:</u> .....	88
<b><u>BUGS IN THE HARDWARE DESIGN</u></b> .....	<b>132</b>
<b><u>POSSIBLE CHANGES IN THE DESIGN</u></b> .....	<b>132</b>
<b><u>REFERENCES</u></b> .....	<b>133</b>

## Development Board 2 - Introduction

The second development board used the 128-pin version of the EZ-USB FX chip. Since this too was meant to be a development board, we brought out all the port, address and data pins as headers along the edges of the board. At each of these headers, VDD and GND pins were provided to make testing an easier task. A CPLD and SRAM were also included in the design. There are two regulators on the board and a provision for an external power supply to be added on (if the application requires more current than the USB port can provide). ATA and CompactFlash interface connectors are also provided to enable connecting this board to a IDE hard drive/ CDROM drive/ CompactFlash card. Some prototyping area is also provided at the bottom of the board enable users to use this board to test and create prototype USB devices. Unlike the first board there are two serial ports, for serial port 0 and serial port 1. Apart from a reset switch the board also has a wake up switch.

### List of switches, jumpers, testing points, connectors used

There are three main options on this board: 1) CompactFlash Interface, 2) ATA interface, 3) CPLD interface. For this project, we only use the CompactFlash interface. In this section, we will briefly explain the pushbuttons, jumpers, testing points, and connectors that are used for the CompactFlash Interface.

#### Pushbuttons

SW1	Reset
Depressed	Board in reset
Open	Not in reset

SW2	Wakeup
Depressed	Active level wakeup interrupt
Open	Inactive level

SW3, SW4	Currently not used
----------	--------------------

SW5, SW6, SW7	Used for CPLD, currently not used
---------------	-----------------------------------

#### Jumpers

J1	Power source
----	--------------

1-2	External power supply
3-4	Bus power supply
J3	Serial communication selector
1-2,5-6	SIO 0 is connected
3-4,7-8	SIO 1 is connected
J4	Used for CPLD, currently not used
J7	Power source
Connected	EEPROM is connected
Open	EEPROM is disconnected
J10	/OE, /WE signal for compactflash
1-2,5-6	Signal is generated from Port E
3-4,7-8	Signal is generated from GPIF
J11, J14	Used for CPLD, currently not used
J18	Secondary Power source
Connected	Extra external power supply
Open	No extra power supply
J19	Power distributor
Connected	Distribute the power when the input power is too high
Open	No distribution
J22, J23	Used for CPLD, currently not used
JP1	Used for CPLD, currently not used
JS-0, JS-1	TXD0, RXD0 selector
3-4,5-6	
1-2,7-8	

#### Testing points

TP1	External power supply
TP2	Vcc
TP3	Gnd
TP4	EEPROM Vcc
TP5	EEPROM Gnd
TP6	CLKOUT
TP7	Compactflash Vcc
TP8	Compactflash Gnd

TP9	Hard drive Vcc
TP10	Hard drive Gnd
TP11	/PSEN
TP12	ADR5
TP13	RDY1
TP14	RDY2
TP15	XCLK
TpinvA, TpinvB, TpinvC, TpinvD	Used for CPLD, currently not used

### Connectors

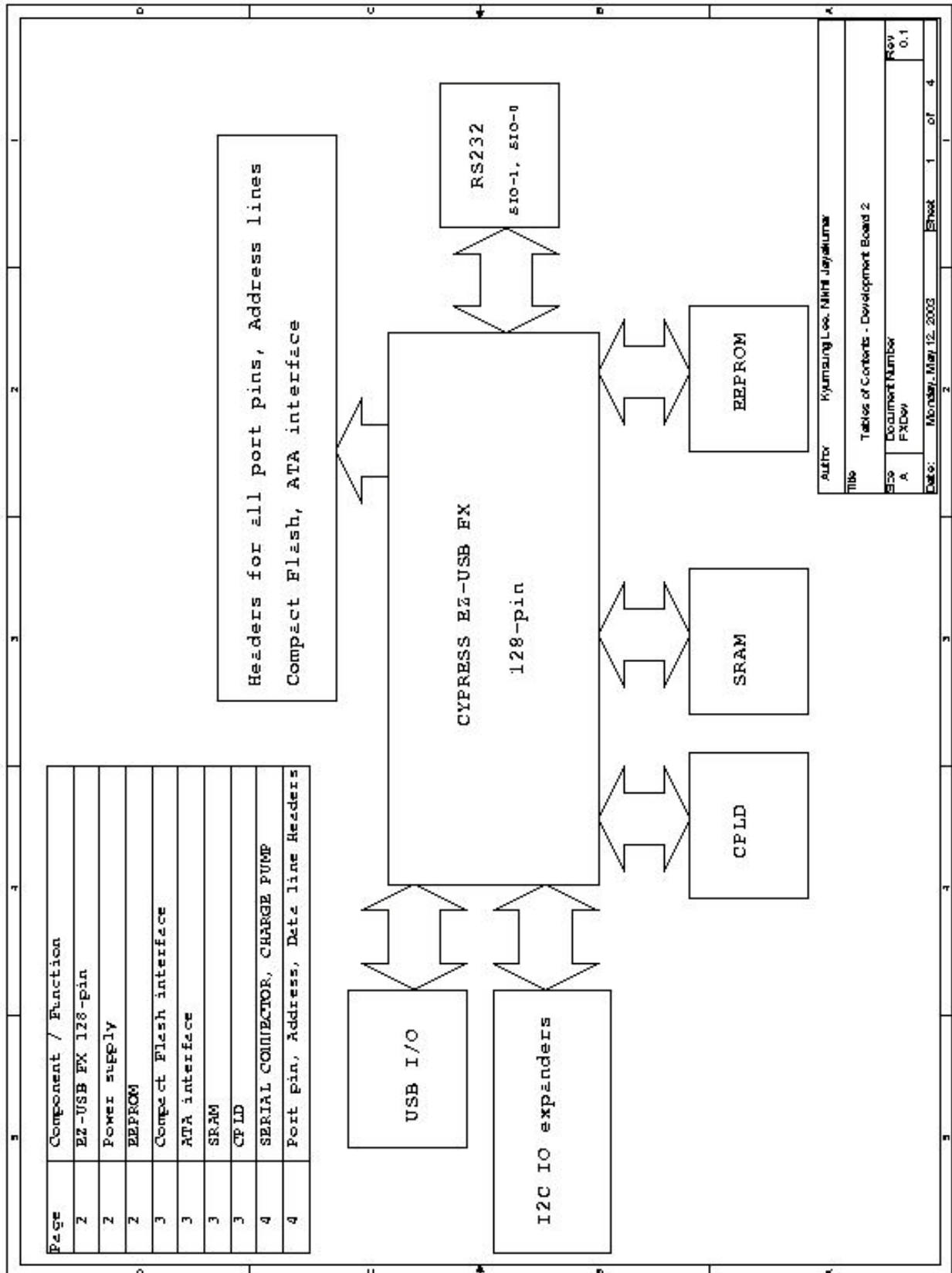
J12	
1	Vcc
3	CFRESET (PA0)
5	/ARESET (PA1)
7	/CE1 (PA2)
9	/CE2 (PA3)
11	CFA0 (PA4)
13	CFA1 (PA5)
15	CFA2 (PA6)
17	CFA3 (PA7)
2	DD0 (PB0)
4	DD1 (PB1)
6	DD2 (PB2)
8	DD3 (PB3)
10	DD4 (PB4)
12	DD5 (PB5)
14	DD6 (PB6)
16	DD7 (PB7)
18	Gnd

J13	
1	Vcc
3	Currently not used
5	Currently not used
7	Currently not used
9	Currently not used
11	Currently not used
13	Currently not used
15	Currently not used
17	Currently not used
2	DD8 (PD0)
4	DD9 (PD1)
6	DD10 (PD2)
8	DD11 (PD3)
10	DD12 (PD4)

12	DD13 (PD5)
14	DD14 (PD6)
16	DD15 (PD7)
18	Gnd

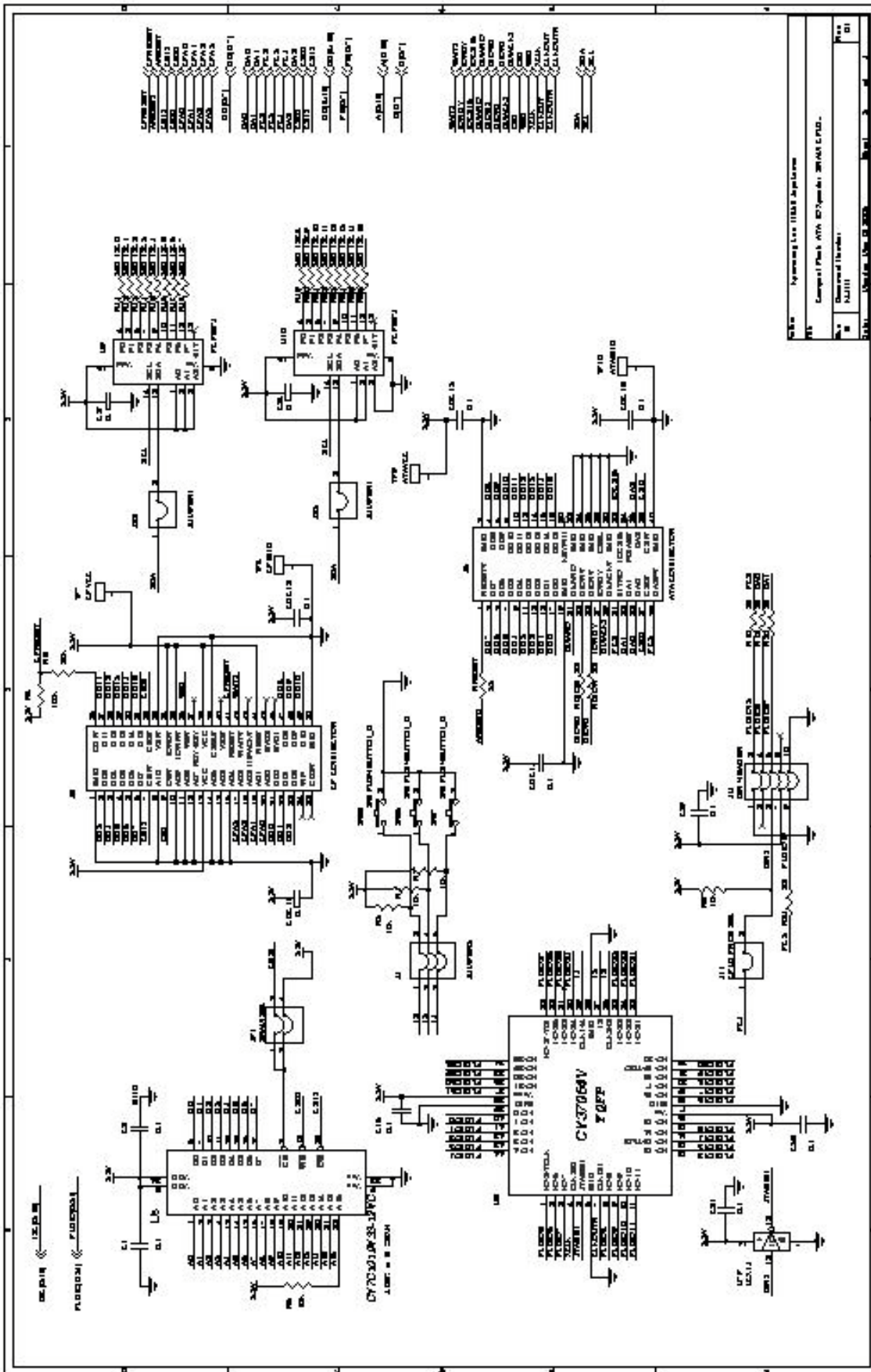
JP15, JP16, JP17, JP20, JP21	Currently not used (Connect to port, address & data pins)
------------------------------	---

## Schematics



Page	Title
1	Tables of Contents - Development Board 2
A	Document Number
FXDev	Rev
	0.1

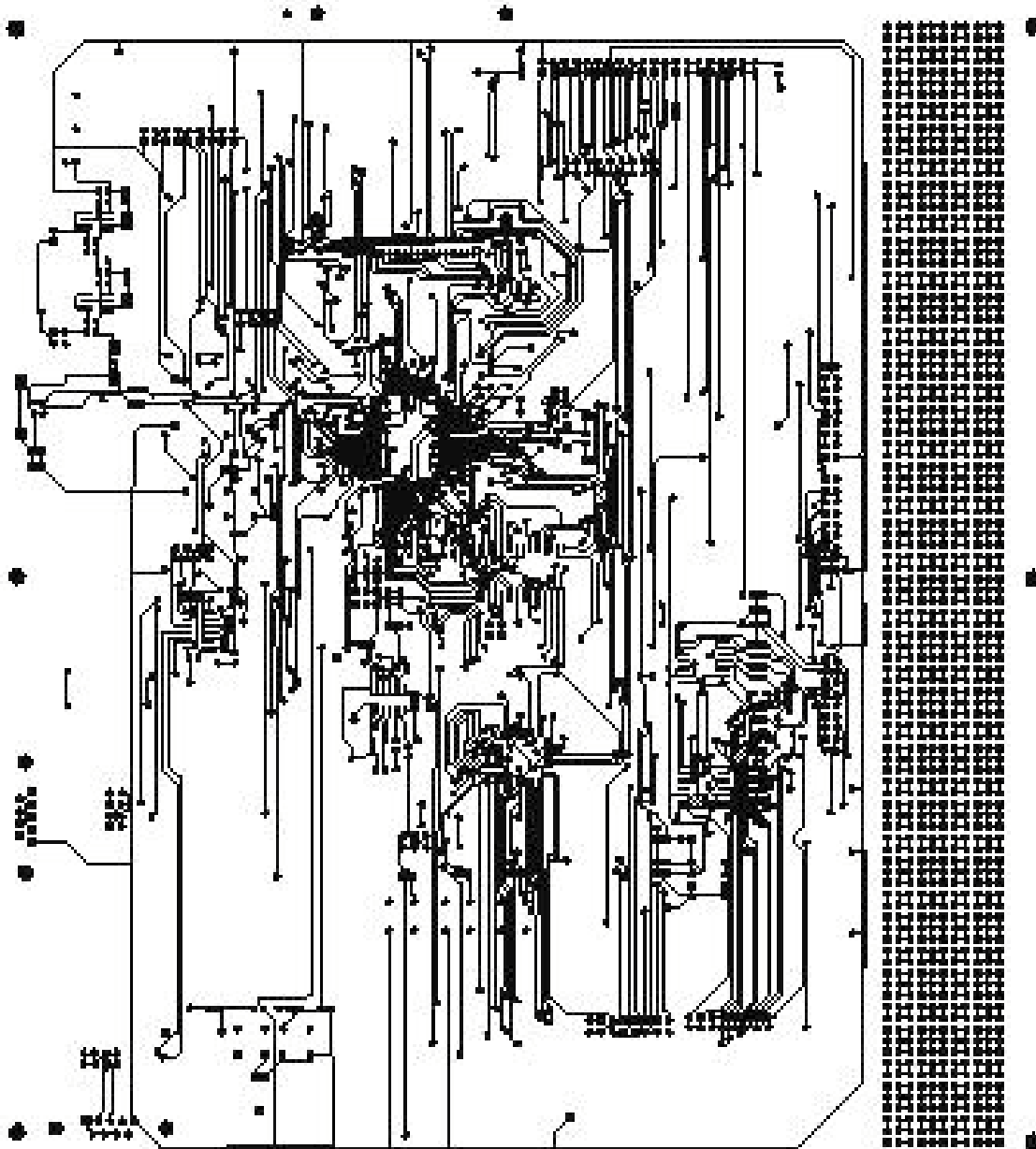




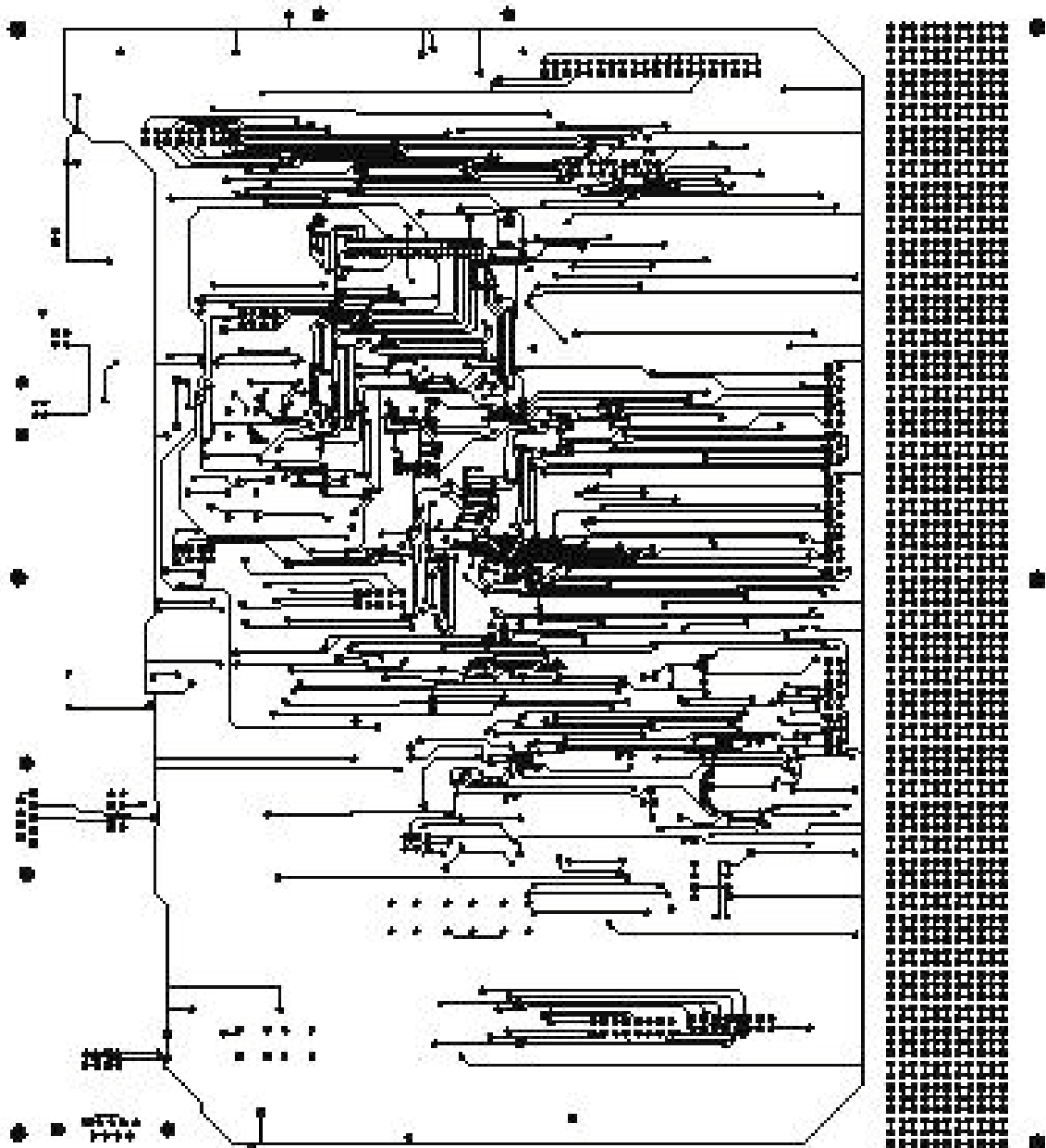




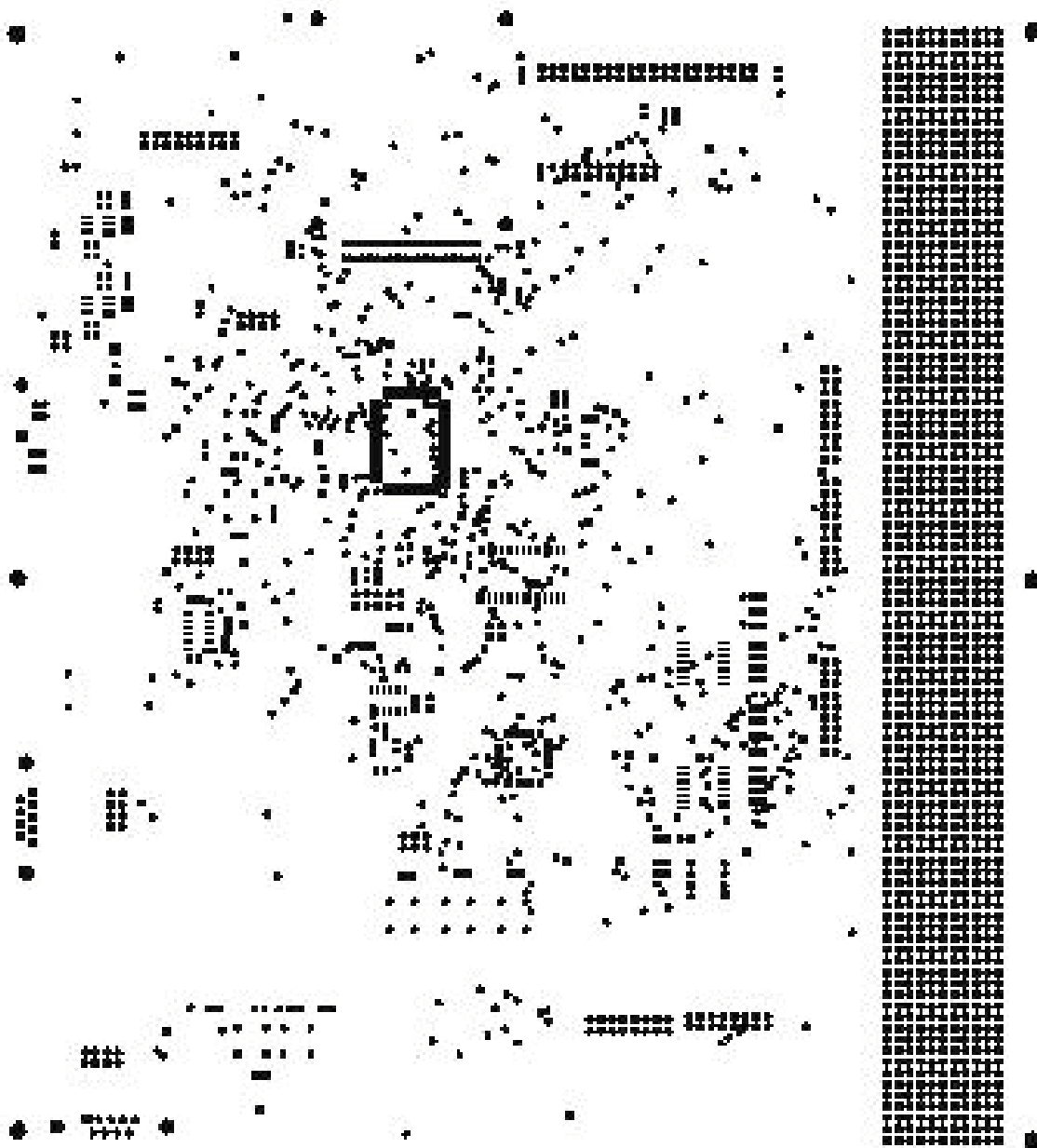
PCB Layout - Top Layer



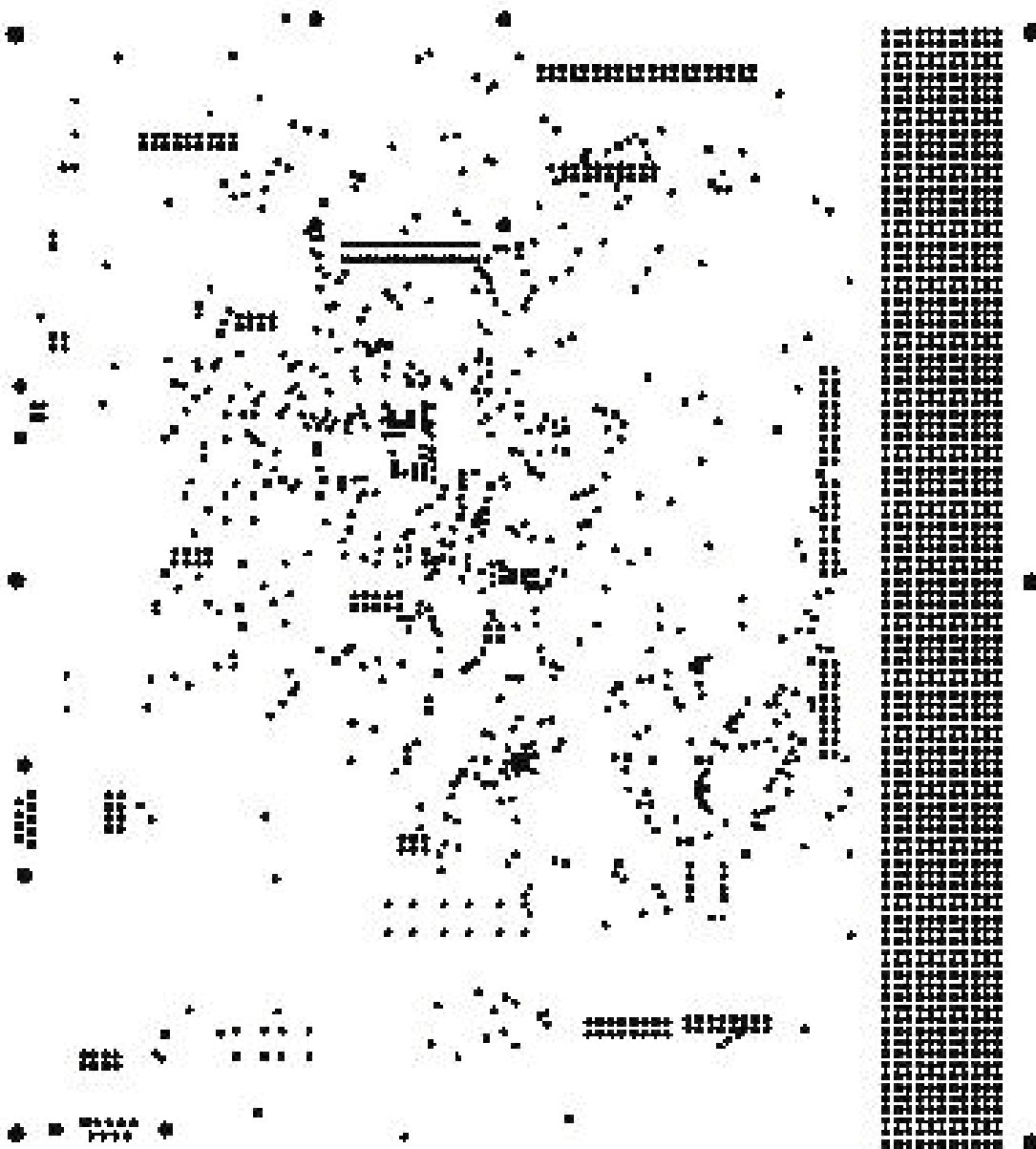
PCB Layout - Bottom Layer



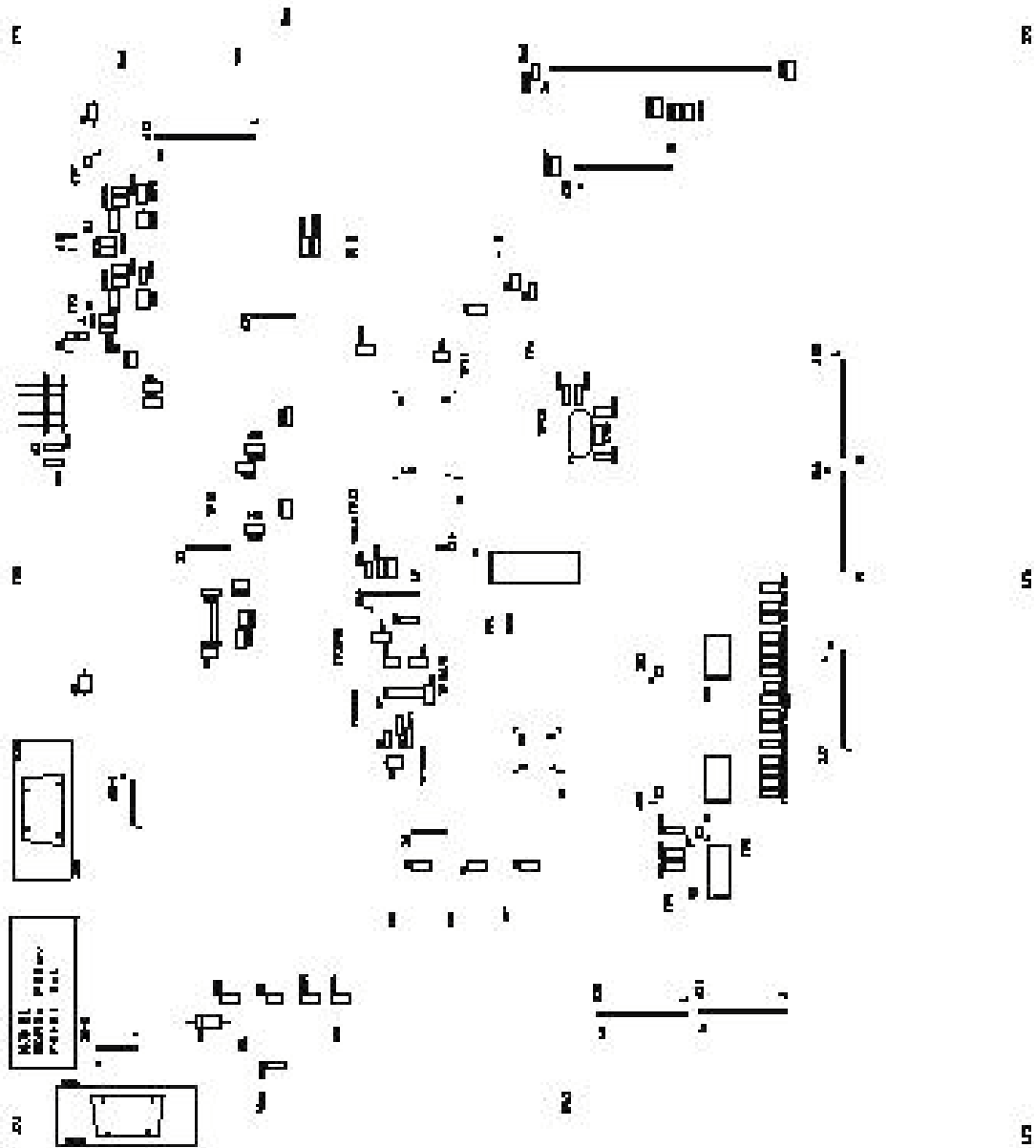
PCB Layout – Solder Mask Top



PCB Layout – Solder Mask Bottom



PCB Layout – Silk Screen Top





## Developing a HID

In developing a USB HID (Human Interface Device), an additional “Report” descriptor has to be added to the file `dscr.a51`. (It is assumed in this document that the user is using the “frameworks” code supplied by Cypress for development). Since we have to add code to handle these additional descriptors, we have to add this code to the function “SetupCommand” (the device request parser) in `fw.c`. For most other applications we wouldn’t change anything in `fw.c`.

### HID Report Descriptor:

One of the requirements of a HID is a report descriptor. Unlike other descriptors, it is not simply a table of values. It is composed of pieces of information called “items” which provide information about the device. When a USB based HID is connected to a computer (running Windows 9x and above), the HID class driver runs a parser that analyzes these items and stores the information contained therein in an item state table. A detailed description of what the contents of a report descriptor should be and in what order the items should be arranged, is given in Section 6.2.2 of the document “USB Device Class Definition for Human Interface Devices” which is downloadable at [http://www.usb.org/developers/devclass\\_docs/HID1\\_11.pdf](http://www.usb.org/developers/devclass_docs/HID1_11.pdf).

A report descriptor can be hard to write out. But this task is made much simpler with a HID Descriptor Tool, which is downloadable at <http://www.usb.org/developers/hidpage/>. This tool allows the developer to create, edit and validate report descriptor tables. There are sample files provided with the tool that help in getting started. The application (`Dt.exe`) is quite intuitive. Existing sample HID report descriptors can be opened and edited or a new report descriptor can be created. To add an item, simply double-click on one of the HID items in the list of HID items displayed. Depending on the type of item, a dialog box should pop-up that allows the user to modify the settings (or values) of the item.

### The Hardware:

The hardware part of the HID essentially consisted of 6 push-button switches that were used to create a pointing device with the ability to left-click and right-click (like a mouse). Four of the switches were arranged to create a joystick-like device. The other two switches were used for the left-click and right-click functions. A Phillips 8-bit I<sup>2</sup>C IO expander was used to receive the inputs from the switches and send the data to the EZ-USB FX chip.

The complete (frameworks) code for USB Human Interface Device:

The “frameworks” code provided by Cypress was used to create the HID. Fw.c contains the main function that calls TD\_Init() from periph.c once and TD\_Poll() (also in periph.c) at regular intervals. Fw.c also contains the function SetupCommand() which is called when a Setup data packet (containing device requests) is received. This function decodes and handles all the device requests. The handles for most device requests were already present in the function, but some additional handles had to be added to the SetupCommand() function to take care of the HID device descriptor, HID report descriptor and HID class requests. Periph.c contains the TD\_Init() function which in this case just initializes end point used (Endpoint 2 IN). A function read\_buttons() was added to read the status of the buttons through the IO expander. This function is called in TD\_Poll(). In TD\_Poll() debouncing of the switch inputs is done and based on the status of the switches, the necessary action (movement / left-click / right-click) is taken. We have also incorporated some code to provide acceleration in the movement of the pointer. The Descriptor tables (including the HID report descriptor) are stored in the assembly file dscr.a51. If Keil’s uVision2 is being used, the library file Ezusb.lib and the file USBImpTb.OBJ, which contains the interrupt re-vectoring information must be added to the source group.

#### Fw.c:

```
//-----  
//      File:          fw.c  
//      Contents:     Firmware frameworks task dispatcher and device request parser  
//                   source.  
//  
//      Copyright (c) 2001 Cypress Semiconductor, Inc. All rights reserved  
//-----  
#include <ezusb.h>  
#include <ezregs.h>  
  
//constants added to handle HID requests  
#define GD_HID 0x21  
#define GD_REPORT 0x22  
#define GD_IF0 0x00  
  
//-----
```

```

// Random Macros
//-----
#define min(a,b) (((a)<(b))?(a):(b))
#define max(a,b) (((a)>(b))?(a):(b))

//-----
// Constants
//-----
#define DELAY_COUNT                0x9248*8L           // Delay for 8 sec at 24Mhz, 4 sec at 48

//-----
// Global Variables
//-----
volatile BOOL    GotSUD;
BOOL            Rwuen;
BOOL            Selfpwr;
volatile BOOL    Sleep;                          // Sleep mode enable flag

WORD pDeviceDscr;    // Pointer to Device Descriptor; Descriptors may be moved
WORD pConfigDscr;
WORD pStringDscr;

//variables added to handle HIDs
WORD pHID1Dscr;
WORD pHID1ReportDscr;
WORD pHID1ReportDscrEnd;
extern code HID1Dscr;
extern code HID1ReportDscr;
extern code HID1ReportDscrEnd;

//-----
// Prototypes
//-----
void SetupCommand(void);
void TD_Init(void);
void TD_Poll(void);
BOOL TD_Suspend(void);
BOOL TD_Resume(void);

BOOL DR_GetDescriptor(void);
BOOL DR_SetConfiguration(void);
BOOL DR_GetConfiguration(void);
BOOL DR_SetInterface(void);

```

```

BOOL DR_GetInterface(void);
BOOL DR_GetStatus(void);
BOOL DR_ClearFeature(void);
BOOL DR_SetFeature(void);
BOOL DR_VendorCmnd(void);
BOOL DR_ClassRequest(void);

//-----
// Code
//-----

// Task dispatcher
void main(void)
{
    DWORD      i;
    WORD  offset;
    DWORD      DevDescrLen;
    DWORD      j=0;
    WORD  IntDescrAddr;
    WORD  ExtDescrAddr;

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

    // Initialize user device
    TD_Init();

    // The following section of code is used to relocate the descriptor table.
    // Since the SUDPTRH and SUDPTRL are assigned the address of the descriptor
    // table, the descriptor table must be located in on-part memory.
    // The 4K demo tools locate all code sections in external memory.
    // The descriptor table is relocated by the frameworks ONLY if it is found
    // to be located in external memory.
    pDeviceDscr = (WORD)&DeviceDscr;
    pConfigDscr = (WORD)&ConfigDscr;
    pStringDscr = (WORD)&StringDscr;
    if ((WORD)&DeviceDscr & 0xe000)
    {
        IntDescrAddr = INTERNAL_DSCR_ADDR;
        ExtDescrAddr = (WORD)&DeviceDscr;
    }
}

```

```

    DevDescrLen = (WORD)&UserDscr - (WORD)&DeviceDscr + 2;
    for (i = 0; i < DevDescrLen; i++)
        *((BYTE xdata *)IntDescrAddr+i) = 0xCD;
    for (i = 0; i < DevDescrLen; i++)
        *((BYTE xdata *)IntDescrAddr+i) = *((BYTE xdata *)ExtDescrAddr+i);
    pDeviceDscr = IntDescrAddr;
    offset = (WORD)&DeviceDscr - INTERNAL_DSCR_ADDR;
    pConfigDscr -= offset;
    pStringDscr -= offset;
}

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

// The 8051 is responsible for all USB events, even those that have happened
// before this point. We cannot ignore pending USB interrupts.
// The chip will come out of reset with the flags all cleared.
//     USBIRQ = 0xFF;                // Clear any pending USB interrupt requests
PORTCCFG |= 0xC0;                // Turn on r/w lines for external memory

USBBAV = USBBAV | 1 & ~bmBREAK; // Disable breakpoints and autovectoring
USBIEN |= bmSUDAV | bmSUTOK | bmSUSP | bmURES; // Enable selected interrupts
EA = 1;                          // Enable 8051 interrupts

#ifdef NO_RENUM
// Note: at full speed, high speed hosts may take 5 sec to detect device
EZUSB_Discon(TRUE); // Renumerate
#endif

CKCON = (CKCON & (~bmSTRETCH)) | FW_STRETCH_VALUE; // Set stretch to 0 (after renumeration)

// Task Dispatcher
while(TRUE)                        // Main Loop
{
    if(GotSUD)                    // Wait for SUDAV
    {
        SetupCommand();           // Implement setup command
        GotSUD = FALSE;          // Clear SUDAV flag
    }

    // Poll User Device
    // 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 other interrupts.
if (Sleep)
{
if(TD_Suspend())
{
Sleep = FALSE; // Clear the "go to sleep" flag. Do it here to prevent any race condition between wakeup and the next sleep.
do
{
EZUSB_Susp(); // Place processor in idle mode.
}
while(!Rwuen && EZUSB_EXTWAKEUP());
// Must continue to go back into suspend if the host has disabled remote wakeup
// *and* the wakeup was caused by the external wakeup pin.

// 8051 activity will resume here due to USB bus or Wakeup# pin activity.
EZUSB_Resume(); // If source is the Wakeup# pin, signal the host to Resume.
TD_Resume();
}
}
TD_Poll();
}
}

// Device request parser
//GD_HID, GD_REPORT , SETUP_CLASS_REQUEST added to handle HID requests
void SetupCommand(void)
{
void *dscr_ptr;
DWORD i;
BYTE HID1length;
switch(SETUPDAT[0] & SETUP_MASK)
{
case SETUP_STANDARD_REQUEST: //Standard Request
switch(SETUPDAT[1])
{
case SC_GET_DESCRIPTOR: // *** Get Descriptor
//following 3 lines added for HIDs
pHID1Dscr = (WORD)&HID1Dscr;
pHID1ReportDscr = (WORD)&HID1ReportDscr;
pHID1ReportDscrEnd = (WORD)&HID1ReportDscrEnd;
if(DR_GetDescriptor())

```

```

switch(SETUPDAT[3])
{
    case GD_DEVICE:                                // Device
        SUDPTRH = MSB(pDeviceDscr);
        SUDPTRL = LSB(pDeviceDscr);
        break;
    case GD_CONFIGURATION:                         // Configuration
        if(dscr_ptr = (void *)EZUSB_GetConfigDscr(SETUPDAT[2]))
        {
            SUDPTRH = MSB(dscr_ptr);
            SUDPTRL = LSB(dscr_ptr);
        }
        else
            EZUSB_STALL_EP0(); // Stall End Point 0
        break;
    case GD_HID:                                  //HID Descriptor
        switch (SETUPDAT[4]) //we use switch-case so that additional interfaces can be easily added
        {
            case GD_IF0:
                SUDPTRH = MSB(pHID1Dscr);
                SUDPTRL = LSB(pHID1Dscr);
                break;
            default:
                EZUSB_STALL_EP0();
        }
        break;
    case GD_REPORT:                               //Report Descriptor
        switch (SETUPDAT[4]) //we use switch-case so that additional interfaces can be easily added
        {
            case GD_IF0:
                HID1length = pHID1ReportDscrEnd - pHID1ReportDscr;
                while (HID1length)
                {
                    for(i=0; i<min(HID1length,64); i++)
                        *(IN0BUF+i) = *((BYTE xdata *)pHID1ReportDscr+i);
                    //set length and arm Endpoint
                    EZUSB_SET_EP_BYTES(IN0BUF_ID,min(HID1length,64));
                    HID1length -= min(HID1length,64);
                    // Wait for it to go out (Rev C and above)
                    while(EP0CS & 0x04)
                        ;
                }
                break;
        }
}

```

```

        default:
            EZUSB_STALL_EP0();
    }
    break;
case GD_STRING: // String
    if(dscr_ptr = (void *)EZUSB_GetStringDscr(SETUPDAT[2]))
    {
        // Workaround for rev D errata number 8
        // If you're certain that you will never run on rev D,
        // you can just do this:
        // SUDPTRH = MSB(dscr_ptr);
        // SUDPTL = LSB(dscr_ptr);
        STRINGDSCR *sdp;
        BYTE len;

        sdp = dscr_ptr;

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

        while (len)
        {
            for(i=0; i<min(len,64); i++)
                *(IN0BUF+i) = *((BYTE xdata *)sdp+i);

            //set length and arm Endpoint
            EZUSB_SET_EP_BYTES(IN0BUF_ID,min(len,64));
            len -= min(len,64);

            // Wait for it to go out (Rev C and above)
            while(EPOCS & 0x04)
                ;
        }

        // Arm a 0 length packet just in case. There was some reflector traffic about
        // Apple hosts asking for too much data. This will keep them happy and will
        // not hurt valid hosts because the next SETUP will clear this.
        EZUSB_SET_EP_BYTES(IN0BUF_ID,0);
        // Clear the HS-nak bit
        EPOCS = bmHS;
    }
else

```

```

        EZUSB_STALL_EP0(); // Stall End Point 0
    break;
    default: // Invalid request
        EZUSB_STALL_EP0(); // Stall End Point 0
    }
    break;
case SC_GET_INTERFACE: // *** Get Interface
    DR_GetInterface();
    break;
case SC_SET_INTERFACE: // *** Set Interface
    DR_SetInterface();
    break;
case SC_SET_CONFIGURATION: // *** Set Configuration
    DR_SetConfiguration();
    break;
case SC_GET_CONFIGURATION: // *** Get Configuration
    DR_GetConfiguration();
    break;
case SC_GET_STATUS: // *** Get Status
    if(DR_GetStatus())
        switch(SETUPDAT[0])
        {
            case GS_DEVICE: // Device
                IN0BUF[0] = ((BYTE)Rwuen << 1) | (BYTE)Selfpwr;
                IN0BUF[1] = 0;
                EZUSB_SET_EP_BYTES(IN0BUF_ID,2);
                break;
            case GS_INTERFACE: // Interface
                IN0BUF[0] = 0;
                IN0BUF[1] = 0;
                EZUSB_SET_EP_BYTES(IN0BUF_ID,2);
                break;
            case GS_ENDPOINT: // End Point
                IN0BUF[0] = EPIO[EPID(SETUPDAT[4]).cntrl & bmEPSTALL];
                IN0BUF[1] = 0;
                EZUSB_SET_EP_BYTES(IN0BUF_ID,2);
                break;
            default: // Invalid Command
                EZUSB_STALL_EP0(); // Stall End Point 0
        }
    break;
case SC_CLEAR_FEATURE: // *** Clear Feature
    if(DR_ClearFeature())

```

```

switch(SETUPDAT[0])
{
    case FT_DEVICE:                                // Device
        if(SETUPDAT[2] == 1)
            Rwuen = FALSE;                        // Disable Remote Wakeup
        else
            EZUSB_STALL_EP0(); // Stall End Point 0
        break;
    case FT_ENDPOINT:                              // End Point
        if(SETUPDAT[2] == 0)
        {
            EZUSB_UNSTALL_EP( EPID(SETUPDAT[4]) );
            EZUSB_RESET_DATA_TOGGLE( SETUPDAT[4] );
        }
        else
            EZUSB_STALL_EP0(); // Stall End Point 0
        break;
    }
    break;
case SC_SET_FEATURE:                              // *** Set Feature
    if(DR_SetFeature())
        switch(SETUPDAT[0])
        {
            case FT_DEVICE:                        // Device
                if(SETUPDAT[2] == 1)
                    Rwuen = TRUE;                // Enable Remote Wakeup
                else
                    EZUSB_STALL_EP0(); // Stall End Point 0
                break;
            case FT_ENDPOINT:                      // End Point
                if(SETUPDAT[2] == 0)
                    EZUSB_STALL_EP( EPID(SETUPDAT[4]) );
                else
                    EZUSB_STALL_EP0(); // Stall End Point 0
                break;
            }
        break;
default:                                          // *** Invalid Command
    EZUSB_STALL_EP0();                          // Stall End Point 0
}
break;
case SETUP_CLASS_REQUEST:                       //Class Request
    if(DR_ClassRequest())

```

```

                EZUSB_STALL_EP0();                // Stall End Point 0
        break;
default:        //Reserved or illegal
                EZUSB_STALL_EP0();                // Stall End Point 0
        break;
}

// Acknowledge handshake phase of device request
// Required for rev C does not effect rev B
EP0CS |= bmBIT1;
}

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

```

---

### periph.c:

```

#pragma NOIV                // Do not generate interrupt vectors
//-----
//      File:                periph.c
//      Contents:           Hooks required to implement USB peripheral function.
//
//      Copyright (c) 2001 Cypress Semiconductor, Inc. All rights reserved
//-----
#include <ezusb.h>
#include <ezregs.h>

#define min(a,b) (((a)<(b))?(a):(b))

#define IOXP_ADDR            0x77 //for reading from PCF8574A, use address 0x47 for PCF8574
#define IOXP_start          0x80
#define IOXP_stop           0x40
#define IOXP_lastrd         0x20
#define IOXP_done           0x01

extern BOOL    GotSUD;                // Received setup data flag
extern BOOL    Sleep;

```

```

void TD_Poll(void);

BYTE Configuration;           // Current configuration
BYTE AlternateSetting; // Alternate settings

BYTE posn;
BYTE oldposn = 0;
BYTE acceln = 0;
BYTE count = 1;
BYTE HID1buttons;
BYTE read_buttons (void);

//-----
// Task Dispatcher hooks
// The following hooks are called by the task dispatcher.
//-----
void TD_Init(void)           // Called once at startup
{
    // Enable endpoint 2 IN

    IN07VAL |= bmEP2; // Validate EP 2 IN
}

BYTE read_buttons (void)
{
    BYTE d;

    while (I2CS & IOXP_stop); //Wait for stop to be done
    I2CS = IOXP_start;        //Set start condition
    I2DAT = IOXP_ADDR;        //Write button address
    while (!(I2CS & IOXP_done)); //Wait for done
    I2CS = IOXP_lastrd;       //Set last read
    d = I2DAT;                //trigger last read
    while (!(I2CS & IOXP_done)); //Wait for done
    I2CS = IOXP_stop;        //Set stop bit
    return(I2DAT);           //Read the data
}

void TD_Poll(void)           // Called repeatedly while the device is idle
{
    if(!(IN2CS & bmEPBUSY)) // Is the IN2BUF available?

```

```

{
HID1buttons = read_buttons();
if (HID1buttons == read_buttons())    // Debounce
{
    HID1buttons &= 0xFF;
    posn = HID1buttons & 0x0F;
    if (oldposn == posn)
        count++;
    else {
        count=1;acceln = 0;
    }
    if (count == 10) {
        acceln++;count = 1;
    }
    IN2BUF[0] = 0x00;           // clear button state as seen by the host
    IN2BUF[1] = 0x00;           // clear button state as seen by the host
    IN2BUF[2] = 0x00;           // clear button state as seen by the host

    if ( !(HID1buttons & 0x01) )    // move right
    {
        IN2BUF[1] |= 0x01 + acceln;
    }

    if ( !(HID1buttons & 0x02) )    // move left
    {
        IN2BUF[1] |= 0xFF - acceln;
    }
    if ( !(HID1buttons & 0x04) )    // move down
    {
        IN2BUF[2] |= 0x01 + acceln;
    }

    if ( !(HID1buttons & 0x08) )    // move up
    {
        IN2BUF[2] |= 0xFF - acceln;
    }
    oldposn = posn;
    if ( !(HID1buttons & 0x10) )    // left click
    {
        IN2BUF[0] |= 0x01;
    }
}

```

```

        if ( !(HID1buttons & 0x80) )    // right click
        {
            IN2BUF[0] |= 0x02;
        }

        IN2BC = 3; //byte count = 3
    }
}

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);
}

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

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

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);
}

//-----
// 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
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUDAV;            // Clear SUDAV IRQ
}

// Setup Token Interrupt Handler
void ISR_Sutok(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUTOK;            // Clear SUTOK IRQ
}

void ISR_Sof(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSOF;              // Clear SOF IRQ
}

void ISR_Ures(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmURES;            // Clear URES IRQ
}

void ISR_IBN(void) interrupt 0
{
    // ISR for the IN Bulk NAK (IBN) interrupt.
}

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

void ISR_Ep0in(void) interrupt 0

```

```
{  
}  
  
void ISR_Ep0out(void) interrupt 0  
{  
}  
  
void ISR_Ep1in(void) interrupt 0  
{  
}  
  
void ISR_Ep1out(void) interrupt 0  
{  
}  
  
void ISR_Ep2in(void) interrupt 0  
{  
}  
  
void ISR_Ep2out(void) interrupt 0  
{  
}  
  
void ISR_Ep3in(void) interrupt 0  
{  
}  
  
void ISR_Ep3out(void) interrupt 0  
{  
}  
  
void ISR_Ep4in(void) interrupt 0  
{  
}  
  
void ISR_Ep4out(void) interrupt 0  
{  
}  
  
void ISR_Ep5in(void) interrupt 0  
{  
}
```

```

void ISR_Ep5out(void) interrupt 0
{
}

void ISR_Ep6in(void) interrupt 0
{
}

void ISR_Ep6out(void) interrupt 0
{
}

void ISR_Ep7in(void) interrupt 0
{
}

void ISR_Ep7out(void) interrupt 0
{
}

```

---

dscr.a51:

```

;-----
;;      File:          dscr.a51
;;      Contents:     This file contains descriptor data for sample mouse/keyboard descriptor tables.
;;
;;      Copyright (c) 2001 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, HID1Dscr, HID1ReportDscr, HID1ReportDscrEnd

```

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 0001H ;; 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 2810H ;; Product ID - set to example ID
            dw 0100H ;; Product version ID
            db 0 ;; Manufacturer string index
            db 0 ;; Product string index
            db 0 ;; Serial number string index
            db 1 ;; Number of configurations
deviceDscrEnd:

ConfigDscr: db ConfigDscrEnd-ConfigDscr ;; Descriptor length
            db DSCR_CONFIG ;; Descriptor type
            db StringDscr-ConfigDscr ;; Configuration + End Points length (LSB)
            db (StringDscr-ConfigDscr)/256 ;; Configuration + End Points length (MSB)
            db 1 ;; Number of interfaces
            db 1 ;; Interface number
            db 0 ;; Configuration string
            db 10100000b ;; Attributes (b7 - buspwr, b6 - selfpwr, b5 - rwu)
            db 0 ;; Power requirement (div 2 ma)
ConfigDscrEnd:

HID1IntrfcDscr:
            db HID1IntrfcDscrEnd-HID1IntrfcDscr ;; Descriptor length
            db DSCR_INTRFC ;; Descriptor type
            db 00H ;; Zero-based index of this interface
            db 0 ;; Alternate setting
            db 1 ;; Number of end points
            db 03H ;; Interface class (HID)
            db 01H ;; Boot Interface sub class
            db 02H ;; Interface sub sub class (Mouse)

```

```

        db      0          ;; Interface descriptor string index
HID1IntrfcDscrEnd:

HID1Dscr:
        db      09h      ; length
        db      21h      ; type: HID
        db      10h,01h   ; release: HID class rev 1.1
        db      00h      ; country code (none)
        db      01h      ; number of HID class descriptors to follow
        db      22h      ; report descriptor type (HID)
        db      (HID1ReportDscrEnd-HID1ReportDscr) ; length of HID descriptor
        db      00h
HID1DscrEnd:

HID1EpInDscr:
        db      HID1EpInDscrEnd-HID1EpInDscr      ;; Descriptor length
        db      DSCR_ENDPNT ;; Descriptor type
        db      82H      ;; Endpoint number, and direction
        db      ET_INT   ;; Endpoint type
        db      40H      ;; Maximum packet size (LSB)
        db      00H      ;; Max packet size (MSB)
        db      10      ;; Polling interval
HID1EpInDscrEnd:

HID1ReportDscr:
        db 05h, 01h ; Usage Page (Generic Desktop),
        db 09h, 02h ; Usage (Mouse),
        db 0A1h, 01h ; Collection (Application),
        db 09h, 01h ; Usage (Pointer),
        db 0A1h, 00h ; Collection (Physical),
        db 95h, 03h ; Report Count (3),
        db 75h, 01h ; Report Size (1),
        db 05h, 09h ; Usage Page (Buttons),
        db 19h, 01h ; Usage minimum (1)
        db 29h, 03h ; Usage maximum (3)
        db 15h, 00h ; Logical minimum (0),
        db 25h, 01h ; Logical maximum (1),
        db 81h, 02h ; Input (Data, Variable, Absolute), (3 button bits)
        db 95h, 01h ; Report Count (1),
        db 75h, 05h ; Report Size (5),
        db 81h, 01h ; Input (Constant)
        db 75h, 08h ; Report Size (8)

```

```

    db 95h, 02h    ;      Report Count (2)
    db 05h, 01h   ;      Usage Page (Generic Desktop),
    db 09h, 30h   ;      Usage (X),
    db 09h, 31h   ;      Usage (Y),
    db 15h, 81h   ;      Logical Minimum (-127),
    db 25h, 7Fh   ;      Logical Maximum (+127),
    db 81h, 06h   ;      Input (Data, Variable, Relative), (2 position bytes - X & Y)
    db 0C0h      ;      End Collection
    db 0C0h      ;      End Collection
HID1ReportDscrEnd:

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      'K',00
    db      'Y',00
    db      'U',00
    db      'M',00
    db      'S',00
    db      'U',00
    db      'N',00
    db      'G',00
    db      '',00
    db      '&',00
    db      ',00
    db      'N',00
    db      'T',00
    db      'K',00
    db      'H',00
    db      'T',00
    db      'L',00
    db      '~',00
    db      'S',00
StringDscr1End:

StringDscr2:

```

```

    db    StringDscr2End-StringDscr2    ;; Descriptor length
    db    DSCR_STRING
    db    'E',00
    db    'Z',00
    db    '-',00
    db    'H',00
    db    'I',00
    db    'D',00
StringDscr2End:
UserDscr:
    dw    0000H
end
```



---

## Misc. Code

Code to Debug Via serial port:

This piece of code can be used to debug (send out information about data received through the USB port) through the serial port on the development board. This is useful when a USB protocol analyzer is not available.

```
#include "ezusb.h"
#include "ezregs.h"
#include "fx.h"

void debugger(BYTE dat);

void main(void)
{
    BYTE dat = 0xB8;
    debugger(dat);
    debugger(0x7a);    //testing 0x7a
    debugger(0xfc);    //testing 0xfc
}

void debugger(BYTE dat)
{
    //IFCONFIG &= 0x00;
    PORTCCCFG |= 0x03;
    PORTCCF2 &= 0xFC;

    TMOD = 0x20;    // Mode 2: 8-bit counter with auto-road
    TCON = 0x40;    // Enable counting on Timer 1
    CKCON = 0x10;    // T1M - Timer 1 uses CLKOUT/12
    SCON0 = 0x50;    // Serial Port 0 - SM mode 1
    TH1 = 0xD9;    // set to 9600 bps
    TL1 = 0x00;
    IE &= 0xF7;    // Disable Timer 1 interrupt
    PCON |= 0x80;    // Serial Port 0 baud rate double enable

    //////////////////////////////////////
    if( (dat >> 4) >=0 && (dat >> 4) <=9)    //higher 4 bits in a byte
```

```

SBUF0 = 0x30 + (dat >> 4);    //displaying 0-9 in ascii
else
SBUF0 = 0x37 + (dat >> 4); //displaying A-F in ascii

    while (!(SCON0 & 0x02))    //waiting for transition end
    ;
    SCON0 &= 0xFD;            //manually clear this bit

if( ((dat & 0x0F) >=0) && ((dat & 0x0F) <=9)) //lower 4 bits in a byte
SBUF0 = 0x30 + (dat & 0x0F); //displaying 0-9 in ascii
else
SBUF0 = 0x37 + (dat & 0x0F); //displaying A-F in ascii

    while (!(SCON0 & 0x02))    //waiting for transition end
    ;
    SCON0 &= 0xFD;            //manually clear this bit

    SBUF0 = 32;                //space
    while (!(SCON0 & 0x02))    //waiting for transition end
    ;
    SCON0 &= 0xFD;            //manually clear this bit
}

```

---

### Compact Flash test:

The following code was used to test if the CompactFlash Interface worked with the GPIF and FIFO Read/Write commands. This code was for the most part generated by the GPIF tool. A few modifications were made since the address lines in our case were from the port-A pins. The main function was also changed to include the test cases. The waveforms used are in the commented out portion of the code (before the functions). The test code below uses functions from Ezusb.lib. If Keil's uVision2 is used, the user must make sure that this library is included in the source group. Note the difference between the different waveforms used for GPIF/FIFO Reads/Writes.

8-bit\_test.c: This file tests 8-bit GPIF read and writes to CompactFlash\_

//Programmer: Kyumsung Lee and Nikhil Jayakumar

```

//Description: This program writes to the compactflash 1 sector
//             and reads back 1 sector with 8 bit data bus.
//             Most of the code is from the cypress control panel.
// This program configures the General Purpose Interface.
// Parts of this program are automatically generated using the GPIF Tool.
// Please do not modify sections of text which are marked as "DO NOT EDIT ...".
// You can modify the comments section of this GPIF program file using the dropdown menus
// and pop-up dialogs. These controls are available as hot spots in the text. Modifying the
// comments section will generate program code which will implement your GPIF program.
//
// DO NOT EDIT ...
// GPIF Initialization
// Interface Timing   Sync
// Data Bus Width    8 BIT           //8 bit test
// Internal Ready Init  IntRdy=1
// CTL Out Tristate-able Binary
// SingleWrite WF Select  3
// SingleRead WF Select  2
// FifoWrite WF Select   1
// FifoRead WF Select    0
// Data Bus Idle Drive  Driven
// END DO NOT EDIT

// DO NOT EDIT ...
// GPIF Wave Names
// Wave 0 = Wave 0
// Wave 1 = Wave 1
// Wave 2 = Wave 2
// Wave 3 = Wave 3

// GPIF Ctrl Outputs Level
// CTL 0 = CTL 0 CMOS
// CTL 1 = CTL 1 CMOS
// CTL 2 = CTL 2 CMOS
// CTL 3 = CTL 3 CMOS
// CTL 4 = CTL 4 CMOS
// CTL 5 = CTL 5 CMOS

// GPIF Rdy Inputs
// RDY0 = RDY0
// RDY1 = RDY1
// RDY2 = RDY2
// RDY3 = RDY3

```

```

// RDY4 = RDY4
// RDY5 = RDY5
// FIFOFlag = FIFOFlag
// IntReady = IntReady
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 0: Wave 0
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____|_____|_____|_____|_____|_____|_____|_____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data NO Data NO Data NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 15 Wait 23 Wait 31 Wait 39 Wait 47 Wait 55 Wait 63
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 1 1 1 1 1 1 1
// CTL 1 1 1 1 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 1: Wave 1
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____|_____|_____|_____|_____|_____|_____|_____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data NO Data NO Data NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int

```

```

// IF/Wait Wait 15 Wait 23 Wait 31 Wait 39 Wait 47 Wait 55 Wait 63
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 1 1 1 1 1 1 1
// CTL 1 1 1 1 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 2: Wave 2
//
// Interval 0 1 2 3 4 5 6 Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 2 Wait 2 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 0 0 1 1 1 1 1
// CTL 1 1 1 1 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT

```

```

// DO NOT EDIT ...
//
// GPIF Waveform 3: Wave 3
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 7 Wait 3 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0  1  1  1  1  1  1  1  1
// CTL 1  1  0  0  1  1  1  1  1
// CTL 2  1  1  1  1  1  1  1  1
// CTL 3  1  1  1  1  1  1  1  1
// CTL 4  1  1  1  1  1  1  1  1
// CTL 5  1  1  1  1  1  1  1  1
//
// END DO NOT EDIT

// GPIF Program Code

// DO NOT EDIT ...
#include "ezusb.h"
#include "ezregs.h"
#include "Fx.h"
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata WaveData[128] =
{
// Wave 0
/* LenBr */ 0x0F, 0x17, 0x1F, 0x27, 0x2F, 0x37, 0x3F, 0x07,
/* Opcode*/ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

```

```

/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 1
/* LenBr */ 0x0F, 0x17, 0x1F, 0x27, 0x2F, 0x37, 0x3F, 0x07,
/* Opcode*/ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 2
/* LenBr */ 0x01, 0x02, 0x02, 0x02, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 3
/* LenBr */ 0x01, 0x02, 0x07, 0x03, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFD, 0xFD, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F
};

```

```
// END DO NOT EDIT
```

```
// DO NOT EDIT ...
```

```
const char xdata InitData[7] =
```

```
{
/* Regs */ 0xC0,0x00,0x00,0xFF,0x02,0xE4,0x00
};
```

```
// END DO NOT EDIT
```

```
// DO NOT EDIT ...
```

```
GpifInit(void)
```

```
{
    unsigned char xdata *Source;
    unsigned char xdata *Dest;
    unsigned int x;
```

```

    ABORT = 0; // abort any pending operation
    READY = InitData[0];
    CTLOUTCFG = InitData[1];
    IDLE_CS = InitData[2];
    IDLE_CTLOUT = InitData[3];
    IFCONFIG = InitData[4];
    WFSELECT = InitData[5];
    ABSETUP |= InitData[6];

```

```

Source = WaveData;           // Transfer the GPIF Tool generated data
Dest = &WFDESC[0];

for (x = 0; x < 128; x++)
    *Dest++ = *Source++;

INT4SETUP = INT4SFC | INT4_INTERNAL; // setup INT4 as internal source
// GENIE = 0x01; // Enable GPIF interrupt
EIEX4 = 0x01;
PORTSETUP |= 0x01;

//PORTCCFG = 0xE0;
PORTACFG = 0x02;
OEA = 0xFD;           //enable Port A
OUTA = 0x0B;         //not reset, /CE1-low, /CE2-low
EZUSB_Delay(10);
OUTA = 0x0A;         //not reset, /CE1-low, /CE2-high
}
// END DO NOT EDIT

// TO DO: You may add additional code below.
//

#define TMOU 0x0020           // Default Timeout TODO: Set this appropriately

void OtherInit()
{
    // TO DO: Add initialization code here.
}

// write byte to PERIPHERAL, using GPIF
bit Peripheral_SingleByteWrite( BYTE gaddr, BYTE gdata )
{
    unsigned char transaction_err = 0x00;

    OUTA = 0x0A | (gaddr*16);           //not reset, /CE1-low, /CE2-high, and control register

    //GPIFADRL = gaddr;           // setup GPIF address ADR0-ADR5
    SGLDATLTRIG = gdata;         // initiate GPIF write transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOU ) // trap GPIF transaction for TMOU

```

```

    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}
return( 1 );
}
/*
// write word to PERIPHERAL, using GPIF
bit Peripheral_SingleWordWrite( BYTE gaddr, WORD gdata )
{
    unsigned char transaction_err = 0x00;

        OEA = 0xFF;
        OUTA = (gaddr<<4) & 0xF0;

//GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
SGLDATH = gdata >> 8;
SGLDATLTRIG = gdata;        // initiate GPIF write transaction

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        {
            ABORT = 0x01;
            return( 0 );          // error has occurred
        }
    }
}

return( 1 );
}
*/
// read byte from PERIPHERAL, using GPIF
bit Peripheral_SingleByteRead( BYTE gaddr, BYTE xdata *gdata )
{
    unsigned char g_data = 0x00;
    unsigned char transaction_err = 0x00;

//OEC = 0x1F;
//OUTC = gaddr & 0x0F;

    OUTA = 0x0A | (gaddr*16);          //not reset, /CE1-low, /CE2-high, and control register

```

```

GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
g_data = SGLDATLTRIG;      // initiate GPIF read transaction

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}

*gdata = SGLDATLNTRIG;

return( 1 );
}
/*
// read word from PERIPHERAL, using GPIF
bit Peripheral_SingleWordRead( BYTE gaddr, WORD xdata *gdata )
{
    unsigned char g_data = 0x00;
    unsigned char transaction_err = 0x00;

        OEA = 0xFF;
        OUTA = (gaddr<<4) & 0xF0;

//GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
g_data = SGLDATLTRIG;      // initiate GPIF read transaction

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}

*gdata = ( ( WORD )SGLDATH << 8 ) | ( WORD )SGLDATLNTRIG;

return( 1 );
}

```

```

// write byte(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOByteWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
    unsigned char transaction_err = 0x00;

    GPIFADRL = gaddr;    // setup GPIF address ADR0-ADR5
    DMASRC = outbuf;    // Typically points to endp buffer
    DMADEST = &AOUTDATA; // point to FIFO-A
    DMALEN = xfrcnt;
    EA = 0;             // protect DMA from interrupts occurring in Block0
    DMAGO = xfrcnt;
    AOUTTC = xfrcnt;   // setup transaction count
    ATRIG = xfrcnt;    // write to ATRIG initiates
                    // FIFO -> GPIF transaction(s)
    while( !( DMAGO & 0x80 ) )
    {
        ; // wait here for the DMA to complete
    }
    EA = 1;             // Enable interrupts...

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        // transaction completed
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 ); // error has occurred
        }
    }

    return( 1 );
}

// write word(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
    unsigned char transaction_err = 0x00;

    OEA = 0xFF;
    OUTA = (gaddr<<4) & 0xF0;

    //GPIFADRL = gaddr;    // setup GPIF address ADR0-ADR5
    DMASRC = outbuf;    // Typically points to endp buffer
    DMADEST = &AOUTDATA; // point to FIFO-A

```

```

DMALEN = xfrcnt;
EA = 0;          // protect DMA from interrupts occurring in Block0
DMAGO = xfrcnt;
AOUTTC = xfrcnt >> 1; // divide by 2 for 16 bit transactions
ATRIG = xfrcnt;    // write to ATRIG initiates
                  // FIFO -> GPIF transaction(s)
while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;          // Enable interrupts...

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    // transaction completed
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 ); // error has occurred
    }
}

return( 1 );
}

// read byte(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOByteRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;
    BYTE gxfr = 0x00;

    GPIFADRL = gaddr; // setup GPIF address ADR0-ADR5
    AINTC = xfrcnt; // setup transaction count
    gxfr = ATRIG; // read from ATRIG initiates
                // GPIF -> FIFO transaction(s)
while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 ); // an error has occurred
    }
}
}

```

```

DMASRC = &AINDATA;          // point to FIFO-A
DMADEST = inbuf;           // Typically points to endp buffer
DMALEN = xfrcnt;

EA = 0;                    // protect DMA from interrupts occurring in Block0
DMAGO = xfrcnt;           // writing any value to DMAGO starts the DMA

while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;                    // Enable interrupts...

return( 1 );
}

// read word(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;
    BYTE gxfr = 0x00;

    OEA = 0xFF;
    OUTA = (gaddr<<4) & 0xF0;

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    AINTC = xfrcnt >> 1;        // divide by 2 for 16 bit interface
    gxfr = ATRIG;              // read from ATRIG initiates
                                // GPIF -> FIFO transaction(s)
    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );           // an error has occurred
        }
    }

    DMASRC = &AINDATA;          // point to FIFO-A
    DMADEST = inbuf;           // Typically points to endp buffer
    DMALEN = xfrcnt;

    EA = 0;                    // protect DMA from interrupts occurring in Block0

```

```

DMAGO = xfrcnt; // writing any value to DMAGO starts the DMA

while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;           // Enable interrupts...

return( 1 );
}
*/

void main( void )
{
    WORD xdata wData;
    BYTE xdata bData;

    BYTE i;
    BYTE m;
    char km;

    bit bResult;

    GpifInit();    //initialize GPIF settings
    OtherInit();

    if(IFCONFIG&0x04) // If 16-bit mode
    { // illustrate use of efficient 16 bit functions
        //bResult = Peripheral_SingleByteWrite(0x00, 0xAA55);
        //bResult = Peripheral_SingleByteRead(0x00, &wData);
    }
    else           // If 8-bit mode
    {

        //////////// write (8 bits) ////////////
        do{
            //Wait_ready
            Peripheral_SingleByteRead(0x07, &bData);
            //if(bData & 0x01)
            //ABORT
        }while((bData & 0xF0) != 0x50);           //wait until compactflash is not busy

        Peripheral_SingleByteWrite(0x02,0x01); //sector count
    }
}

```

```
Peripheral_SingleByteWrite(0x03,0x06); //LBA 7-0
Peripheral_SingleByteWrite(0x04,0x30); //LBA 15-8
Peripheral_SingleByteWrite(0x05,0x00); //LBA 23-16
Peripheral_SingleByteWrite(0x06,0xE0); //Drive/Head
Peripheral_SingleByteWrite(0x07,0x30); //Write Command
```

```
do{ //Wait for DRQ
    Peripheral_SingleByteRead(0x07, &bData);
    //if(bData & 0x10)
        //ABORT
}while((bData & 0xF8) != 0x58); //wait until compactflash is ready
```

//Compactflash needs to be written or read 512 bytes at one time.

```
for(i=0; i<128; i++)
    Peripheral_SingleByteWrite(0x00,i); //write 128 bytes
```

```
for(i=128; i>0; i--)
    Peripheral_SingleByteWrite(0x00,i); //write 128 bytes
```

```
for(i=0; i<128; i++)
    Peripheral_SingleByteWrite(0x00,i); //write 128 bytes
```

```
for(i=64; i>0; i--)
    Peripheral_SingleByteWrite(0x00,i); //write 64 bytes
```

```
Peripheral_SingleByteWrite(0x00,0xFF); //write 64 bytes
Peripheral_SingleByteWrite(0x00,0xEE);
Peripheral_SingleByteWrite(0x00,0xDD);
Peripheral_SingleByteWrite(0x00,0xCC);
Peripheral_SingleByteWrite(0x00,0xBB);
Peripheral_SingleByteWrite(0x00,0xAA);
Peripheral_SingleByteWrite(0x00,0x99);
Peripheral_SingleByteWrite(0x00,0x32);
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x34); //10
```

```
Peripheral_SingleByteWrite(0x00,0x31);
Peripheral_SingleByteWrite(0x00,0x32);
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x34);
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x36);
```

```
Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x38);
Peripheral_SingleByteWrite(0x00,0x39);
Peripheral_SingleByteWrite(0x00,0x38); //20
```

```
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x36);
Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x38);
Peripheral_SingleByteWrite(0x00,0x39);
Peripheral_SingleByteWrite(0x00,0x38);
Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x36);
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x34); //30
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x32);
Peripheral_SingleByteWrite(0x00,0x31);
```

```
Peripheral_SingleByteWrite(0x00,0x32);
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x34);
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x36);
Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x38); //40
Peripheral_SingleByteWrite(0x00,0x39);
Peripheral_SingleByteWrite(0x00,0x38);
Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x36);
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x34);
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x32);
Peripheral_SingleByteWrite(0x00,0x31);
```

```
Peripheral_SingleByteWrite(0x00,0x32); //50
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x34);
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x36);
Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x38);
Peripheral_SingleByteWrite(0x00,0x39);
Peripheral_SingleByteWrite(0x00,0x38);
```

```

Peripheral_SingleByteWrite(0x00,0x37);
Peripheral_SingleByteWrite(0x00,0x36); //60
Peripheral_SingleByteWrite(0x00,0x35);
Peripheral_SingleByteWrite(0x00,0x34);
Peripheral_SingleByteWrite(0x00,0x33);
Peripheral_SingleByteWrite(0x00,0x32);

```

```

////////// read (8 bits) //////////
do{
    //Wait_ready
    Peripheral_SingleByteRead(0x07, &bData);
    //if(bData & 0x10)
    //ABORT
}while((bData & 0xF0) != 0x50); //wait until compactflash is not busy

Peripheral_SingleByteWrite(0x02,0x01); //sector count
Peripheral_SingleByteWrite(0x03,0x06); //LBA 7-0
Peripheral_SingleByteWrite(0x04,0x30); //LBA 15-8
Peripheral_SingleByteWrite(0x05,0x00); //LBA 23-16
Peripheral_SingleByteWrite(0x06,0xE0); //Drive/Head
Peripheral_SingleByteWrite(0x07,0x20); //Write Command

do{
    //Wait for DRQ
    Peripheral_SingleByteRead(0x07, &bData);
    //if(bData & 0x10)
    //ABORT
}while((bData & 0xF8) != 0x58); //wait until compactflash is ready

for(i=0; i<128; i++){
    Peripheral_SingleByteRead(0x00,&bData); //read 128 bytes
} //128

for(i=0; i<64; i++){
    Peripheral_SingleByteRead(0x00,&bData); //read 64 bytes
}

for(i=0; i<64; i++){
    Peripheral_SingleByteRead(0x00,&bData); //read 64 bytes
    IN2BUF[i]=bData; //store Endpoint2 IN buffer
} //256

for(i=0; i<64; i++){
    Peripheral_SingleByteRead(0x00,&bData); //read 64 bytes
}

```

```

    }

    for(i=0; i<64; i++){
        Peripheral_SingleByteRead(0x00,&bData); //read 64 bytes
        IN4BUF[i]=bData; //store Endpoint4 IN buffer
    } //384

    for(i=0; i<64; i++){
        Peripheral_SingleByteRead(0x00,&bData); //read 64 bytes
    }

    for(i=0; i<64; i++){
        Peripheral_SingleByteRead(0x00,&bData); //read 64 bytes
        IN6BUF[i]=bData; //store Endpoint6 IN buffer
    } //512

    IN2BC = 64; // arm the transfer
    IN4BC = 64; // arm the transfer
    IN6BC = 64; // arm the transfer
}
while(1);
}

```

---

### 16bittest.c: This code tests 16-bit GPIF Reads and Writes to CompactFlash

```

//Programmer: Kyumsung Lee and Nikhil Jayakumar
//Description: This program writes to the compactflash 1 sector
//             and reads back 1 sector with 8 bit data bus.
//             Most of the code is from the cypress control panel.
// This program configures the General Purpose Interface.
// Parts of this program are automatically generated using the GPIF Tool.
// Please do not modify sections of text which are marked as "DO NOT EDIT ...".
// You can modify the comments section of this GPIF program file using the dropdown menus
// and pop-up dialogs. These controls are available as hot spots in the text. Modifying the
// comments section will generate program code which will implement your GPIF program.
//
// DO NOT EDIT ...
// GPIF Initialization
// Interface Timing    Sync
// Data Bus Width     16 BIT
// Internal Ready Init IntRdy=1

```

```

// CTL Out Tristate-able Binary
// SingleWrite WF Select 3
// SingleRead WF Select 2
// FifoWrite WF Select 1
// FifoRead WF Select 0
// Data Bus Idle Drive Tristate
// END DO NOT EDIT

// DO NOT EDIT ...
// GPIF Wave Names
// Wave 0 = Wave 0
// Wave 1 = Wave 1
// Wave 2 = Wave 2
// Wave 3 = Wave 3

// GPIF Ctrl Outputs Level
// CTL 0 = CTL 0 CMOS
// CTL 1 = CTL 1 CMOS
// CTL 2 = CTL 2 CMOS
// CTL 3 = CTL 3 CMOS
// CTL 4 = CTL 4 CMOS
// CTL 5 = CTL 5 CMOS

// GPIF Rdy Inputs
// RDY0 = RDY0
// RDY1 = RDY1
// RDY2 = RDY2
// RDY3 = RDY3
// RDY4 = RDY4
// RDY5 = RDY5
// FIFOFlag = FIFOFlag
// IntReady = IntReady
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 0: Wave 0
//
// Interval 0 1 2 3 4 5 6 Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data NO Data NO Data NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData

```

```

// Int Trig No Int  No Int  No Int  No Int  No Int  No Int  No Int
// IF/Wait Wait 15  Wait 23  Wait 31  Wait 39  Wait 47  Wait 55  Wait 63
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default  Default  Default  Default  Default  Default  Default
// CTL 0   1   1   1   1   1   1   1   1
// CTL 1   1   1   1   1   1   1   1   1
// CTL 2   1   1   1   1   1   1   1   1
// CTL 3   1   1   1   1   1   1   1   1
// CTL 4   1   1   1   1   1   1   1   1
// CTL 5   1   1   1   1   1   1   1   1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 1: Wave 1
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____|_____|_____|_____|_____|_____|_____|_____
//
// AddrMode Same Val  Same Val  Same Val  Same Val  Same Val  Same Val  Same Val
// DataMode NO Data  NO Data  NO Data  NO Data  NO Data  NO Data  NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int  No Int  No Int  No Int  No Int  No Int
// IF/Wait Wait 15  Wait 23  Wait 31  Wait 39  Wait 47  Wait 55  Wait 63
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default  Default  Default  Default  Default  Default  Default
// CTL 0   1   1   1   1   1   1   1   1
// CTL 1   1   1   1   1   1   1   1   1
// CTL 2   1   1   1   1   1   1   1   1
// CTL 3   1   1   1   1   1   1   1   1
// CTL 4   1   1   1   1   1   1   1   1
// CTL 5   1   1   1   1   1   1   1   1
//

```

```

// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 2: Wave 2
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 2 Wait 2 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0  1  0  0  1  1  1  1  1
// CTL 1  1  1  1  1  1  1  1  1
// CTL 2  1  1  1  1  1  1  1  1
// CTL 3  1  1  1  1  1  1  1  1
// CTL 4  1  1  1  1  1  1  1  1
// CTL 5  1  1  1  1  1  1  1  1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 3: Wave 3
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 7 Wait 3 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B

```

```

// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 1 1 1 1 1 1 1
// CTL 1 1 0 0 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT

// GPIF Program Code

// DO NOT EDIT ...
#include "ezusb.h"
#include "ezregs.h"
#include "Fx.h"
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata WaveData[128] =
{
// Wave 0
/* LenBr */ 0x0F, 0x17, 0x1F, 0x27, 0x2F, 0x37, 0x3F, 0x07,
/* Opcode*/ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 1
/* LenBr */ 0x0F, 0x17, 0x1F, 0x27, 0x2F, 0x37, 0x3F, 0x07,
/* Opcode*/ 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 2
/* LenBr */ 0x01, 0x02, 0x02, 0x02, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 3
/* LenBr */ 0x01, 0x02, 0x07, 0x03, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFD, 0xFD, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

```

```

/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F
};
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata InitData[7] =
{
/* Regs */ 0xC0,0x00,0x00,0xFF,0x06,0xE4,0x11
};
// END DO NOT EDIT

// DO NOT EDIT ...
GpifInit(void)
{
    unsigned char xdata *Source;
    unsigned char xdata *Dest;
    unsigned int x;

    ABORT = 0;                // abort any pending operation
    READY = InitData[0];
    CTLOUTCFG = InitData[1];
    IDLE_CS = InitData[2];
    IDLE_CTLOUT = InitData[3];
    IFCONFIG = InitData[4];
    WFSELECT = InitData[5];
    ABSETUP |= InitData[6];

    Source = WaveData;       // Transfer the GPIF Tool generated data
    Dest = &WFDESC[0];

    for (x = 0; x < 128; x++)
        *Dest++ = *Source++;

    INT4SETUP = INT4SFC | INT4_INTERNAL; // setup INT4 as internal source
// GENIE = 0x01; // Enable GPIF interrupt
    EIEX4 = 0x01;
    PORTSETUP |= 0x01;

    //PORTCCFG = 0xE0;
    PORTACFG = 0x02;
    OEA = 0xFD;              //enable Port A
    OUTA = 0x03;            //not reset, /CE1-low, /CE2-low
    EZUSB_Delay(10);
}

```

```

        OUTA = 0x02;          //not reset, /CE1-low, /CE2-low
    }
// END DO NOT EDIT

// TO DO: You may add additional code below.
//

#define TMOU 0x0020          // Default Timeout TODO: Set this appropriately

void OtherInit()
{
    // TO DO: Add initialization code here.
}

// write byte to PERIPHERAL, using GPIF
bit Peripheral_SingleByteWrite( BYTE gaddr, BYTE gdata )
{
    unsigned char transaction_err = 0x00;

    OUTA = 0x0A | (gaddr*16);    //not reset, /CE1-low, /CE2-high, and control register

    GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    SGLDATLTRIG = gdata;       // initiate GPIF write transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOU ) // trap GPIF transaction for TMOU
        {
            ABORT = 0x01;
            return( 0 );          // error has occurred
        }
    }
    return( 1 );
}

// write word to PERIPHERAL, using GPIF
bit Peripheral_SingleWordWrite( BYTE gaddr, WORD gdata )
{
    unsigned char transaction_err = 0x00;

    OUTA = 0x02 | (gaddr*16);    //not reset, /CE1-low, /CE2-low, and control register

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5

```

```

SGLDATH = gdata >> 8;
SGLDATLTRIG = gdata;          // initiate GPIF write transaction

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}

return( 1 );
}

// read byte from PERIPHERAL, using GPIF
bit Peripheral_SingleByteRead( BYTE gaddr, BYTE xdata *gdata )
{
    unsigned char g_data = 0x00;
    unsigned char transaction_err = 0x00;

    OUTA = 0x0A | (gaddr*16); //not reset, /CE1-low, /CE2-high, and control register

    GPIFADRL = gaddr;        // setup GPIF address ADR0-ADR5
    g_data = SGLDATLTRIG;    // initiate GPIF read transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );          // error has occurred
        }
    }

    *gdata = SGLDATLNTRIG;

    return( 1 );
}

// read word from PERIPHERAL, using GPIF
bit Peripheral_SingleWordRead( BYTE gaddr, WORD xdata *gdata )

```

```

{
  unsigned char g_data = 0x00;
  unsigned char transaction_err = 0x00;

  OUTA = 0x02 | (gaddr*16);          //not reset, /CE1-low, /CE2-low, and control register

  //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
  g_data = SGLDATLTRIG;        // initiate GPIF read transaction

  while( !( IDLE_CS & 0x80 ) )    // poll IDLE_CS.7 Done bit
  {
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
      ABORT = 0x01;
      return( 0 );          // error has occurred
    }
  }

  *gdata = ( ( WORD )SGLDATH << 8 ) | ( WORD )SGLDATLNTRIG;

  return( 1 );
}
/*
// write byte(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOByteWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
  unsigned char transaction_err = 0x00;

  GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
  DMASRC = outbuf;          // Typically points to endp buffer
  DMADEST = &AOUTDATA; // point to FIFO-A
  DMALEN = xfrcnt;
  EA = 0;          // protect DMA from interrupts occurring in Block0
  DMAGO = xfrcnt;
  AOUTTC = xfrcnt;    // setup transaction count
  ATRIG = xfrcnt;    // write to ATRIG initiates
                  // FIFO -> GPIF transaction(s)
  while( !( DMAGO & 0x80 ) )
  {
    ; // wait here for the DMA to complete
  }
  EA = 1;          // Enable interrupts...

```

```

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    // transaction completed
    if( ++transaction_err > TMOUT) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 ); // error has occurred
    }
}

return( 1 );
}

// write word(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
    unsigned char transaction_err = 0x00;

    OEA = 0xFF;
    OUTA = (gaddr<<4) & 0xF0;

    //GPIFADRL = gaddr; // setup GPIF address ADR0-ADR5
    DMASRC = outbuf; // Typically points to endp buffer
    DMADEST = &AOUTDATA; // point to FIFO-A
    DMALEN = xfrcnt;
    EA = 0; // protect DMA from interrupts occurring in Block0
    DMAGO = xfrcnt;
    AOUTTC = xfrcnt >> 1; // divide by 2 for 16 bit transactions
    ATRIG = xfrcnt; // write to ATRIG initiates
    // FIFO -> GPIF transaction(s)
    while( !( DMAGO & 0x80 ) )
    {
        ; // wait here for the DMA to complete
    }
    EA = 1; // Enable interrupts...

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        // transaction completed
        if( ++transaction_err > TMOUT) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 ); // error has occurred
        }
    }
}

```

```

return( 1 );
}

// read byte(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOByteRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;
    BYTE gxfr = 0x00;

    GPIFADRL = gaddr;           // setup GPIF address ADR0-ADR5
    AINTC = xfrcnt;             // setup transaction count
    gxfr = ATRIG;               // read from ATRIG initiates
                                // GPIF -> FIFO transaction(s)
    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );           // an error has occurred
        }
    }

    DMASRC = &AINDATA;         // point to FIFO-A
    DMADEST = inbuf;           // Typically points to endp buffer
    DMALEN = xfrcnt;

    EA = 0;                     // protect DMA from interrupts occurring in Block0
    DMAGO = xfrcnt;             // writing any value to DMAGO starts the DMA

    while( !( DMAGO & 0x80 ) )
    {
        ; // wait here for the DMA to complete
    }
    EA = 1;                       // Enable interrupts...

    return( 1 );
}

// read word(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;

```

```

BYTE gxfr = 0x00;

    OEA = 0xFF;
    OUTA = (gaddr<<4) & 0xF0;

//GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
AINTC = xfrcnt >> 1;        // divide by 2 for 16 bit interface
gxfr = ATRIG;               // read from ATRIG initiates
                            // GPIF -> FIFO transaction(s)
while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );           // an error has occurred
    }
}

DMASRC = &AINDATA;         // point to FIFO-A
DMADEST = inbuf;           // Typically points to endp buffer
DMALEN = xfrcnt;

EA = 0;                    // protect DMA from interrupts occurring in Block0
DMAGO = xfrcnt;           // writing any value to DMAGO starts the DMA

while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;                    // Enable interrupts...

return( 1 );
}
*/

void main( void )
{
    WORD xdata wData;
    BYTE xdata bData;

    BYTE i;
    WORD m;
    char km;

```

bit bResult;

```
GpifInit();  
OtherInit();
```

```
////////// write (16 bits) //////////  
do{  
    //Wait_ready  
    Peripheral_SingleByteRead(0x07, &bData);  
    //if(wData & 0x01)  
    //ABORT  
}while((bData & 0xF0) != 0x50);    //wait until compactflash is not busy  
  
Peripheral_SingleByteWrite(0x02,0x01); //sector count  
Peripheral_SingleByteWrite(0x03,0x08); //LBA 7-0  
Peripheral_SingleByteWrite(0x04,0x30); //LBA 15-8  
Peripheral_SingleByteWrite(0x05,0x00); //LBA 23-16  
Peripheral_SingleByteWrite(0x06,0xE0); //Drive/Head  
Peripheral_SingleByteWrite(0x07,0x30); //Write Command  
  
do{ //Wait for DRQ  
    Peripheral_SingleByteRead(0x07, &bData);  
    //if(wData & 0x10)  
    //ABORT  
}while((bData & 0xF8) != 0x58);    //wait until compactflash is ready  
  
m = 0x12f0;  
for(i=0; i<128; i++){  
    Peripheral_SingleWordWrite(0x00,m); //write 256 bytes  
    m++;  
}  
  
m = 0x3100;  
for(i=0; i<64; i++){  
    Peripheral_SingleWordWrite(0x00,m); //write 128 bytes  
    m++;  
}  
  
m = 0x2200;  
for(i=0; i<64; i++){  
    Peripheral_SingleWordWrite(0x00,m); //write 128 bytes  
    m++;
```

```

}

////////// read (16 bits) //////////
do{
    //Wait_ready
    Peripheral_SingleByteRead(0x07, &bData);
    //if(wData & 0x10)
    //ABORT
}while((bData & 0xF0) != 0x50);    //wait until compactflash is not busy

Peripheral_SingleByteWrite(0x02,0x01); //sector count
Peripheral_SingleByteWrite(0x03,0x08); //LBA 7-0
Peripheral_SingleByteWrite(0x04,0x30); //LBA 15-8
Peripheral_SingleByteWrite(0x05,0x00); //LBA 23-16
Peripheral_SingleByteWrite(0x06,0xE0); //Drive/Head
Peripheral_SingleByteWrite(0x07,0x20); //Read Command

do{
    //Wait for DRQ
    Peripheral_SingleByteRead(0x07, &bData);
    //if(wData & 0x10)
    //ABORT
}while((bData & 0xF8) != 0x58);    //wait until compactflash is ready

for(i=0; i<64; i++){
    Peripheral_SingleWordRead(0x00,&wData);    //read 128 bytes
    IN2BUF[i]=wData;    //store Endpoint2 IN buffer
} //64

for(i=0; i<128; i++){
    Peripheral_SingleWordRead(0x00,&wData);    //read 256 bytes
} //256

for(i=0; i<64; i++){
    Peripheral_SingleWordRead(0x00,&wData);    //read 128 bytes
    IN6BUF[i]=wData;    //lower byte, store Endpoint4 IN buffer
    IN4BUF[i]= wData >> 8;    //higher byte, store Endpoint6 IN buffer
} //256

IN2BC = 64;    // arm the transfer
IN4BC = 64;    // arm the transfer
IN6BC = 64;    // arm the transfer

while(1);
}

```

---

## fifotest.c: This code tests 16-bit FIFO Reads and Writes

```
//Programmer: Kyumsung Lee and Nikhil Jayakumar
//Description: This program writes to the compactflash 1 sector
//             and reads back 1 sector with 8 bit data bus.
//             Most of the code is from the cypress control panel.
// This program configures the General Purpose Interface.
// Parts of this program are automatically generated using the GPIF Tool.
// Please do not modify sections of text which are marked as "DO NOT EDIT ...".
// You can modify the comments section of this GPIF program file using the dropdown menus
// and pop-up dialogs. These controls are available as hot spots in the text. Modifying the
// comments section will generate program code which will implement your GPIF program.
//
// DO NOT EDIT ...
// GPIF Initialization
// Interface Timing    Sync
// Data Bus Width     16 BIT
// Internal Ready Init  IntRdy=1
// CTL Out Tristate-able Binary
// SingleWrite WF Select  3
// SingleRead WF Select  2
// FifoWrite WF Select   1
// FifoRead WF Select    0
// Data Bus Idle Drive  Tristate
// END DO NOT EDIT

// DO NOT EDIT ...
// GPIF Wave Names
// Wave 0 = Wave 0
// Wave 1 = Wave 1
// Wave 2 = Wave 2
// Wave 3 = Wave 3

// GPIF Ctrl Outputs Level
// CTL 0 = CTL 0  CMOS
// CTL 1 = CTL 1  CMOS
// CTL 2 = CTL 2  CMOS
// CTL 3 = CTL 3  CMOS
// CTL 4 = CTL 4  CMOS
// CTL 5 = CTL 5  CMOS
```

```

// GPIF Rdy Inputs
// RDY0 = RDY0
// RDY1 = RDY1
// RDY2 = RDY2
// RDY3 = RDY3
// RDY4 = RDY4
// RDY5 = RDY5
// FIFOFlag = FIFOFlag
// IntReady = IntReady
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 0: Wave 0
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData NextData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 2 Wait 2 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0  1  0  0  1  1  1  1  1
// CTL 1  1  1  1  1  1  1  1  1
// CTL 2  1  1  1  1  1  1  1  1
// CTL 3  1  1  1  1  1  1  1  1
// CTL 4  1  1  1  1  1  1  1  1
// CTL 5  1  1  1  1  1  1  1  1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 1: Wave 1
//
// Interval  0    1    2    3    4    5    6  Idle (7)

```

```

// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 7 Wait 3 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 1 1 1 1 1 1 1
// CTL 1 1 0 0 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 2: Wave 2
//
// Interval 0 1 2 3 4 5 6 Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 2 Wait 2 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 0 0 1 1 1 1 1
// CTL 1 1 1 1 1 1 1 1 1

```

```

// CTL 2   1   1   1   1   1   1   1   1
// CTL 3   1   1   1   1   1   1   1   1
// CTL 4   1   1   1   1   1   1   1   1
// CTL 5   1   1   1   1   1   1   1   1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 3: Wave 3
//
// Interval  0   1   2   3   4   5   6   Idle (7)
// _____|_____|_____|_____|_____|_____|_____|_____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 7 Wait 3 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0   1   1   1   1   1   1   1   1
// CTL 1   1   0   0   1   1   1   1   1
// CTL 2   1   1   1   1   1   1   1   1
// CTL 3   1   1   1   1   1   1   1   1
// CTL 4   1   1   1   1   1   1   1   1
// CTL 5   1   1   1   1   1   1   1   1
//
// END DO NOT EDIT

// GPIF Program Code

// DO NOT EDIT ...
#include "ezusb.h"
#include "ezregs.h"
#include "Fx.h"
// END DO NOT EDIT

// DO NOT EDIT ...

```

```

const char xdata WaveData[128] =
{
// Wave 0
/* LenBr */ 0x01, 0x02, 0x02, 0x02, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x04, 0x00,
/* Output*/ 0xFF, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 1
/* LenBr */ 0x01, 0x02, 0x07, 0x03, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x04, 0x00,
/* Output*/ 0xFF, 0xFD, 0xFD, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 2
/* LenBr */ 0x01, 0x02, 0x02, 0x02, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 3
/* LenBr */ 0x01, 0x02, 0x07, 0x03, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFD, 0xFD, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F
};
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata InitData[7] =
{
/* Regs */ 0xC0,0x00,0x00,0xFF,0x06,0xE4,0x11
};
// END DO NOT EDIT

// DO NOT EDIT ...
GpifInit(void)
{
    unsigned char xdata *Source;
    unsigned char xdata *Dest;
    unsigned int x;

    ABORT = 0; // abort any pending operation
    READY = InitData[0];
    CTLOUTCFG = InitData[1];
    IDLE_CS = InitData[2];

```

```

IDLE_CTLOUT = InitData[3];
IFCONFIG = InitData[4];
WFSELECT = InitData[5];
ABSETUP |= InitData[6];

Source = WaveData;           // Transfer the GPIF Tool generated data
Dest = &WFDESC[0];

for (x = 0; x < 128; x++)
    *Dest++ = *Source++;

INT4SETUP = INT4SFC | INT4_INTERNAL; // setup INT4 as internal source
// GENIE = 0x01; // Enable GPIF interrupt
EIEX4 = 0x01;
PORTSETUP |= 0x01;

//PORTCCFG = 0xE0;
PORTACFG = 0x02;
OEA = 0xFD;           //enable Port A
OUTA = 0x03;         //not reset, /CE1-low, /CE2-low
EZUSB_Delay(10);
OUTA = 0x02;         //not reset, /CE1-low, /CE2-low
}
// END DO NOT EDIT

// TO DO: You may add additional code below.
//

#define TMOUT 0x0020           // Default Timeout TODO: Set this appropriately

void OtherInit()
{
    // TO DO: Add initialization code here.
}

// write byte to PERIPHERAL, using GPIF
bit Peripheral_SingleByteWrite( BYTE gaddr, BYTE gdata )
{
    unsigned char transaction_err = 0x00;

    OUTA = 0x0A | (gaddr*16);           //not reset, /CE1-low, /CE2-high, and control register

    GPIFADRL = gaddr;                 // setup GPIF address ADR0-ADR5

```

```

SGLDATLTRIG = gdata;          // initiate GPIF write transaction

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}
return( 1 );
}
/*
// write word to PERIPHERAL, using GPIF
bit Peripheral_SingleWordWrite( BYTE gaddr, WORD gdata )
{
    unsigned char transaction_err = 0x00;

    OUTA = 0x02 | (gaddr*16);          //not reset, /CE1-low, /CE2-low, and control register

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    SGLDATH = gdata >> 8;
    SGLDATLTRIG = gdata;          // initiate GPIF write transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );          // error has occurred
        }
    }

    return( 1 );
}
*/
// read byte from PERIPHERAL, using GPIF
bit Peripheral_SingleByteRead( BYTE gaddr, BYTE xdata *gdata )
{
    unsigned char g_data = 0x00;
    unsigned char transaction_err = 0x00;

    OUTA = 0x0A | (gaddr*16);          //not reset, /CE1-low, /CE2-high, and control register

```

```

GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
g_data = SGLDATLTRIG;      // initiate GPIF read transaction

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}

*gdata = SGLDATLNTRIG;

return( 1 );
}
/*
// read word from PERIPHERAL, using GPIF
bit Peripheral_SingleWordRead( BYTE gaddr, WORD xdata *gdata )
{
    unsigned char g_data = 0x00;
    unsigned char transaction_err = 0x00;

    OUTA = 0x02 | (gaddr*16); //not reset, /CE1-low, /CE2-low, and control register

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    g_data = SGLDATLTRIG;      // initiate GPIF read transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );          // error has occurred
        }
    }

    *gdata = ( ( WORD )SGLDATH << 8 ) | ( ( WORD )SGLDATLNTRIG;

    return( 1 );
}

```

```

// write byte(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOByteWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
    unsigned char transaction_err = 0x00;

    GPIFADRL = gaddr;    // setup GPIF address ADR0-ADR5
    DMASRC = outbuf;    // Typically points to endp buffer
    DMADEST = &AOUTDATA; // point to FIFO-A
    DMALEN = xfrcnt;
    EA = 0;             // protect DMA from interrupts occurring in Block0
    DMAGO = xfrcnt;
    AOUTTC = xfrcnt;    // setup transaction count
    ATRIG = xfrcnt;    // write to ATRIG initiates
                    // FIFO -> GPIF transaction(s)
    while( !( DMAGO & 0x80 ) )
    {
        ; // wait here for the DMA to complete
    }
    EA = 1;            // Enable interrupts...

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        // transaction completed
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 ); // error has occurred
        }
    }

    return( 1 );
}
*/
// write word(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
    unsigned char transaction_err = 0x00;

    OUTA = 0x02 | (gaddr*16); //not reset, /CE1-low, /CE2-low, and control register

    //GPIFADRL = gaddr;    // setup GPIF address ADR0-ADR5
    DMASRC = outbuf;    // Typically points to endp buffer
    DMADEST = &AOUTDATA; // point to FIFO-A
    DMALEN = xfrcnt;

```

```

EA = 0;          // protect DMA from interrupts occurring in Block0
DMAGO = xfrcnt;
AOUTTC = xfrcnt >> 1; // divide by 2 for 16 bit transactions
ATRIG = xfrcnt;    // write to ATRIG initiates
                // FIFO -> GPIF transaction(s)
while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;          // Enable interrupts...

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    // transaction completed
    if( ++transaction_err > TMOUT) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // error has occurred
    }
}

return( 1 );
}
/*
// read byte(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOByteRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;
    BYTE gxfr = 0x00;

    GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    AINTC = xfrcnt;           // setup transaction count
    gxfr = ATRIG;             // read from ATRIG initiates
    // GPIF -> FIFO transaction(s)
while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
{
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );          // an error has occurred
    }
}

DMASRC = &AINDATA;          // point to FIFO-A

```

```

DMADEST = inbuf;          // Typically points to endp buffer
DMALEN = xfrcnt;

EA = 0;          // protect DMA from interrupts occurring in Block0
DMAGO = xfrcnt; // writing any value to DMAGO starts the DMA

while( !( DMAGO & 0x80 ) )
{
    ;// wait here for the DMA to complete
}
EA = 1;          // Enable interrupts...

return( 1 );
}
*/
// read word(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;
    BYTE gxfr = 0x00;

    OUTA = 0x02 | (gaddr*16);    //not reset, /CE1-low, /CE2-low, and control register

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    AINTC = xfrcnt >> 1;        // divide by 2 for 16 bit interface
    gxfr = ATRIG;              // read from ATRIG initiates
                                // GPIF -> FIFO transaction(s)
    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );          // an error has occurred
        }
    }

    DMASRC = &AINDATA;         // point to FIFO-A
    DMADEST = inbuf;          // Typically points to endp buffer
    DMALEN = xfrcnt;

    EA = 0;          // protect DMA from interrupts occurring in Block0
    DMAGO = xfrcnt; // writing any value to DMAGO starts the DMA

```

```

while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;           // Enable interrupts...

return( 1 );
}

void main( void )
{
    WORD xdata wData;
    BYTE xdata bData;

    BYTE i;
    BYTE m;
    char km;

    bit bResult;

    GpifInit();
    OtherInit();

    //////////// write (16 bits) ////////////
    do{           //Wait_ready
        Peripheral_SingleByteRead(0x07, &bData);
        //if(wData & 0x01)
            //ABORT
    }while((bData & 0xF0) != 0x50);           //wait until compactflash is not busy

    Peripheral_SingleByteWrite(0x02,0x01); //sector count
    Peripheral_SingleByteWrite(0x03,0x0a); //LBA 7-0
    Peripheral_SingleByteWrite(0x04,0x30); //LBA 15-8
    Peripheral_SingleByteWrite(0x05,0x00); //LBA 23-16
    Peripheral_SingleByteWrite(0x06,0xE0); //Drive/Head
    Peripheral_SingleByteWrite(0x07,0x30); //Write Command

    do{           //Wait for DRQ
        Peripheral_SingleByteRead(0x07, &bData);
        //if(wData & 0x10)
            //ABORT
    }while((bData & 0xF8) != 0x58);           //wait until compactflash is ready

```

```

/*
m = 0x0600;
for(i=0; i<64; i++){
    Peripheral_SingleWordWrite(0x00,m);    //write 64 bytes, writing to Endpoint 2 OUT buffer
    m++;
}

m = 0x1300;
for(i=0; i<64; i++){
    Peripheral_SingleWordWrite(0x00,m);    //write 64 bytes, writing to Endpoint 2 OUT buffer
    m++;
}

m = 0x2400;
for(i=0; i<64; i++){
    Peripheral_SingleWordWrite(0x00,m);    //write 64 bytes, writing to Endpoint 2 OUT buffer
    m++;
}

m = 0x3700;
for(i=0; i<64; i++){
    Peripheral_SingleWordWrite(0x00,m);    //write 64 bytes, writing to Endpoint 2 OUT buffer
    m++;
}
*/

m = 0x12;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    IN6BUF[i]=m;
    m++;
} //64
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0x38;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    m++;
} //128
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0xa3;
for(i=0; i<64; i++){

```

```

        OUT2BUF[i]=m;
        m++;
    } //192
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0x98;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    m++;
} //256
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0x45;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    m++;
} //320
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0x11;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    m++;
} //384
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0xd1;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    m++;
} //448
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

m = 0x24;
for(i=0; i<64; i++){
    OUT2BUF[i]=m;
    m++;
} //512
Peripheral_AFIFOWordWrite(0x00,64,OUT2BUF); //write 64 bytes, writing to Endpoint 2 OUT buffer

```

//////////////////////////////// read (16 bits) //////////////////////////////////

```

do{          //Wait_ready
Peripheral_SingleByteRead(0x07, &bData);
//if(wData & 0x10)
//ABORT
}while((bData & 0xF0) != 0x50);          //wait until compactflash is not busy

Peripheral_SingleByteWrite(0x02,0x01); //sector count
Peripheral_SingleByteWrite(0x03,0x0a); //LBA 7-0
Peripheral_SingleByteWrite(0x04,0x30); //LBA 15-8
Peripheral_SingleByteWrite(0x05,0x00); //LBA 23-16
Peripheral_SingleByteWrite(0x06,0xE0); //Drive/Head
Peripheral_SingleByteWrite(0x07,0x20); //Read Command

do{        //Wait for DRQ
Peripheral_SingleByteRead(0x07, &bData);
//if(wData & 0x10)
//ABORT
}while((bData & 0xF8) != 0x58);          //wait until compactflash is ready

Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN2BUF); //read 64 bytes, store Endpoint2 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN4BUF); //read 64 bytes, store Endpoint4 IN buffer
Peripheral_AFIFOWordRead(0x00,64,IN6BUF); //read 64 bytes, store Endpoint6 IN buffer

IN2BC = 64; // arm the transfer
IN4BC = 64; // arm the transfer
IN6BC = 64; // arm the transfer

while(1);    }

```

The complete (frameworks) code for USB Compact Flash Mass Storage Device:

Here we used the frameworks code to make a Mass Storage Device using Compact Flash media. Fw.c contains the main function that calls TD\_Init() from periph.c once and TD\_Poll() (also in periph.c) at regular intervals. Fw.c also contains the function SetupCommand() which is called when a Setup data packet (containing device requests) is received. This function decodes and handles all the device requests.

Periph.c contains the TD\_Init() function which initializes the Compact Flash hardware interface and sets up the GPIF. TD\_Poll() checks up the contents of the bulk output endpoint buffer (in our case the bulk output endpoint 2) and if there is something processes it (extracts the instruction from the data). The Reduced Block Command (RBC) set is used and the data is processed based on this fact. Depending on the instruction it is decided if it is an OUT type of instruction (Read, Verify, Read capacity..) or an IN type of instruction (Write). Accordingly the functions that do the Read (ideReadCommand) or Write (ideWriteCommand) are called. These functions (which we added to periph.c) make calls to the GPIF/FIFO reads and writes (in the file gpif1.c) which actually carry out the Reading and Writing from/to the Compact Flash card. The Descriptor tables are stored in the assembly file dscr.a51. If Keil's uVision2 is being used, the library file Ezusb.lib and the file USBImpTb.OBJ which contains the interrupt re-vectoring information must be added to the source group.

However, so far we have not been able to get the software to work properly. The host does manage to identify that this a USB Mass Storage Device, but after that it does not find the appropriate driver. That is why there might be some parts of the code below which are commented out, but not removed (since we are still in the process of debugging it).

Some pieces of the code are from a reference design for a EZ-USB FX IDE/ATAPI Mass storage Device. Most of the significant portions of it though, are from the frameworks code.

Fw.c:

```
//-----  
// 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 "fx.h"  
#include "gpif.h"  
#include "periph.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 1

//-----
// Global Variables
//-----
volatile BOOL  GotSUD ;
BOOL          Rwuen  ;
BOOL          Selfpwr ;
volatile BOOL  Sleep ;           // Sleep mode enable flag
BYTE  Configuration; // Current configuration
BYTE  AlternateSetting; // Alternate settings

WORD  pDeviceDscr; // Pointers to Descriptors; Descriptors may be moved
WORD  pConfigDscr;
WORD  pStringDscr;

//-----
// 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;           // Initialize sleep flag
    Rwuon = FALSE;          // Disable remote wakeup
    Selfpwr = TRUE;         // Disable self powered
    GotSUD = FALSE;         // Clear "Got setup data" flag

// #if GPIF
// Check to see if VBUS is connected. If not, wait disconnected until it is.
// while (!(PINSC & PORTC_GPIF_VBUS_DETECT_MASK))
//     OUTA &= ~GPIF_DISCON_; // Disconnect via port pin
// #endif

// Check to see if we have been initialized before. If so, it's just a USB reset. Don't disconnect.
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();

    pDeviceDscr = (WORD)&DeviceDscr;
    pConfigDscr = (WORD)&ConfigDscr;
    pStringDscr = (WORD)&StringDscr;

// Warning -- If you are NOT using the disconnect pin, you will want to move TD_Init to execute
// immediately before the task dispatcher. In addition, you will want to move all of the interrupt init stuff
// into your usb reset processing routine. You will NOT want to do this here, as a SETUP packet could come
// from the host between reset and this line. That message would be lost at this point.
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
                    SUDPTRH = MSB(&ConfigDscr);
                    SUDPTL = LSB(&ConfigDscr);
                    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++)
    {
        IN0BUF[i] = ((BYTE xdata *)sdp)[i];
    }
    EZUSB_SET_EP_BYTES(IN0BUF_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
    IN0BUF[0] = AlternateSetting;
    EZUSB_SET_EP_BYTES(IN0BUF_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
    IN0BUF[0] = Configuration;
    EZUSB_SET_EP_BYTES(IN0BUF_ID,1);
    break;
case SC_GET_STATUS: // *** Get Status
    if(DR_GetStatus())
        switch(setupCopy[0])

```

```

{
case GS_DEVICE:           // Device
    IN0BUF[0] = ((BYTE)Rwuen << 1) | (BYTE)Selfpwr;
    IN0BUF[1] = 0;
    EZUSB_SET_EP_BYTES(IN0BUF_ID,2);
    break;
case GS_INTERFACE:       // Interface
    IN0BUF[0] = 0;
    IN0BUF[1] = 0;
    EZUSB_SET_EP_BYTES(IN0BUF_ID,2);
    break;
case GS_ENDPOINT:       // End Point
    IN0BUF[0] = EPIO[epid(SetupCopy[4]).cntrl & ~bmEPBUSY];
    IN0BUF[1] = 0;
    EZUSB_SET_EP_BYTES(IN0BUF_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));
}

```

---

#### periph.c:

```

#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
//-----
#include "ezusb.h"
#include "ezregs.h"
#include "fx.h"
#include "CFconst.h"
#include "gpif.h"
// #include "ide.h"

void mymemmovexx(BYTE xdata * dest, BYTE xdata * src, BYTE len);
void processCBW();
void sendUSBS(BYTE passOrFail);
void failedIn();
bit waitForBusyBit(WORD timeout);
bit generalIDEInCommand();
bit generalIDEOutCommand();
bit waitForInBuffer();
bit ideReadCommand(bit verify);
bit ideWriteCommand();
//bit ideReadCommandTest();
//bit ideWriteCommandTest();

```

```

void IDEnop();
BYTE getStatus(BYTE gaddr);
//void readPIO16toXdata(char addr, char xdata *inbuffer, WORD count);
//void ATAPIIdDevice();
bit checkForMedia();
void hardwareReset();
void initCF();

extern void SetupCommand(void);
extern BOOL    GotSUD;                // Received setup data flag
extern BOOL    Sleep;
extern BOOL    Rwen;
extern BOOL    Selfpwr;

static bit prevMediaStatus;
extern BYTE    Configuration;        // Current configuration
extern BYTE    AlternateSetting;    // Alternate settings

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

const char code senseInvalidFieldInCDB[] = {0x70, 0x00, 0x05, 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};
const char code senseNoMedia[] = {0x70, 0x00, 0x02, 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, 0x28, 0x00, 0x00, 0x00, 0x00, 0x00};
char code * sensePtr;

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, //
0x43, 0x79, 0x70, 0x72, 0x65, 0x73, 0x73, 0x20, // = Manufacturer "Cypress "
0x41, 0x54, 0x41, 0x50, 0x49, 0x20, 0x52, 0x65, 0x66, 0x20, 0x44, 0x65, 0x73, 0x69, 0x67, 0x6e, // = 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
{
    Rwuen = TRUE;                // Enable remote-wakeup
    GpifInit();
    initCF();
    // 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;
    checkForMedia();
    EZUSB_Delay(500);           // Wait 500 ms to make sure it's really ready.
    //ATAPIIdDevice();         // Get serial number
}

char const code usbcString[] = "USBC"; //dCBWsignature
void TD_Poll(void)                // Called repeatedly while the device is idle
{
    BYTE byteCount;
    if( !(EPIO[OUT2BUF_ID].cntrl & bmEPBUSY) ) // Is there something in the OUT2BUF buffer,
    {
        // Check for dCBWsignature
        if ( *((DWORD xdata *)OUT2BUF) != *((DWORD xdata *) usbcString))
        {
            EZUSB_STALL_EP(OUT2BUF_ID); //Bad Pkt, stall OUT2 EP
        }
        else
        {
            byteCount = EPIO[OUT2BUF_ID].bytes;
            if (byteCount < OUT2BUF[CBW_CBW_LEN]+CBW_DATA_START)
                EZUSB_STALL_EP(OUT2BUF_ID); //Bad Pkt, stall OUT2 EP
            else

```

```

        processCBW();// Good packet, forward to the device.
    }
}

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)
    {
        sendUSBS(generalIDEInCommand());
    }
    else
    {
        sendUSBS(generalIDEOutCommand());
    }
}

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

void sendUSBS(BYTE passOrFail)
{
    while (EPIO[IN2BUF_ID].cntrl & (bmEPBUSY | bmEPSTALL)); // Wait for an available buffer
    // Fill the buffer & send the data back to the host
    IN2BUF[0] = 'U';
    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];
IN2BUF[8+1] = ((BYTE *)&dataTransferLen)[2];
IN2BUF[8+2] = ((BYTE *)&dataTransferLen)[1];
IN2BUF[8+3] = ((BYTE *)&dataTransferLen)[0];

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

void failedIn()
{
    // Stall if the host is still expecting data
    if (dataTransferLen)
    {
        EZUSB_Delay(2);          // Wait 2 ms to make sure the endpoint is clear
        EZUSB_STALL_EP(IN2BUF_ID);
    }
}

bit waitForBusyBit(WORD timeout)
{
    BYTE driveStatus;

    do
    {
        driveStatus = getStatus(CF_STATUS_REG);
    }
    while((driveStatus & (CF_STATUS_BUSY_BIT)) && --timeout); // Do-while

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

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);
}

```

```

}

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

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);
}

//-----
// 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(); // Implement setup command
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUDAV; // Clear SUDAV IRQ
}

// Setup Token Interrupt Handler
void ISR_Sutok(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSUTOK; // Clear SUTOK IRQ
}

void ISR_Sof(void) interrupt 0
{
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmSOF; // Clear SOF IRQ
}

void ISR_Ures(void) interrupt 0

```

```

{
    EZUSB_IRQ_CLEAR();
    USBIRQ = bmURES;                // Clear URES IRQ

    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;

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

void ISR_IBN(void) interrupt 0
{
}

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

void ISR_Ep0in(void) interrupt 0
{
}

void ISR_Ep0out(void) interrupt 0
{
}

```

```
void ISR_Ep1in(void) interrupt 0
{
}

void ISR_Ep1out(void) interrupt 0
{
}

void ISR_Ep2in(void) interrupt 0
{
}

void ISR_Ep2out(void) interrupt 0
{
}

void ISR_Ep3in(void) interrupt 0
{
}

void ISR_Ep3out(void) interrupt 0
{
}

void ISR_Ep4in(void) interrupt 0
{
}

void ISR_Ep4out(void) interrupt 0
{
}

void ISR_Ep5in(void) interrupt 0
{
}

void ISR_Ep5out(void) interrupt 0
{
}

void ISR_Ep6in(void) interrupt 0
{
```

```

}

void ISR_Ep6out(void) interrupt 0
{
}

void ISR_Ep7in(void) interrupt 0
{
}

void ISR_Ep7out(void) interrupt 0
{
}

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(0)); // ideReadCommand sets the OUT2BC = 0
    else
    {
        EPIO[OUT2BUF_ID].bytes = 0;    // relinquish control of the bulk buffer occupied by the CBW
        failedIn();
        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 CBW
        failedIn();
        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)[1];
        IN2BUF[1] = ((BYTE *) &driveCapacity)[0];
        IN2BUF[2] = ((BYTE *) &driveCapacity)[3];
        IN2BUF[3] = ((BYTE *) &driveCapacity)[2];
        IN2BUF[4] = (CF_SECTOR_SIZE >> 24) & 0xff;
        IN2BUF[5] = (CF_SECTOR_SIZE >> 16) & 0xff;
        IN2BUF[7] = (CF_SECTOR_SIZE >> 0) & 0xff;
        IN2BUF[6] = (CF_SECTOR_SIZE >> 8) & 0xff;
        IN2BC = sizeof(DWORD) * 2;
        status = USBS_PASSED;
    }

```

```

else
{
failedIn();
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();
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();
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();
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 CBW
                return(USBS_FAILED);
            }

        default:
            EPIO[OUT2BUF_ID].bytes = 0;    // relinquish control of the bulk buffer occupied by the CBW
            if (dataTransferLen)
                EZUSB_STALL_EP(OUT2BUF_ID); //stall OUT2 EP
            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;
    //bit chk;

```

```

    BYTE chkbyte;
    DWORD dwLBA;
    Peripheral_SingleByteWrite( CF_DRIVESEL_REG, 0xe0 );
    //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);
    Peripheral_SingleByteWrite(CF_DRIVESEL_REG, 0xe0);
    Peripheral_SingleByteWrite(CF_LBA_LSB_REG, ((char *) &dwLBA)[3]);
    Peripheral_SingleByteWrite(CF_LBA_2SB_REG, ((char *) &dwLBA)[2]);
    Peripheral_SingleByteWrite(CF_LBA_MSB_REG, ((char *) &dwLBA)[1]);
    Peripheral_SingleByteWrite(CF_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
        Peripheral_SingleByteWrite(CF_DRIVESEL_REG, 0xe0);
        Peripheral_SingleByteWrite(CF_LBA_LSB_REG, ((char *) &dwLBA)[3]);
        Peripheral_SingleByteWrite(CF_LBA_2SB_REG, ((char *) &dwLBA)[2]);
        Peripheral_SingleByteWrite(CF_LBA_MSB_REG, ((char *) &dwLBA)[1]);
        Peripheral_SingleByteWrite(CF_DRIVESEL_REG, ((char *) &dwLBA)[0] | 0xe0);

        // First stuff the length register (number of sectors to read)
        if (dataTransferLenMSW & 0xfffe)
        {
            Peripheral_SingleByteWrite(CF_SECTOR_COUNT_REG, 0);    // 0 means 256 blocks of 512
            sectorcount = 0x100;    //max sectorcount =256, i.e. 256 sectors
        }
    }

```

```

    }
else
{
    sectorcount = ((dataTransferLenLSW)/(CF_SECTOR_SIZE)) + (dataTransferLenMSW & 1) * 0x80;
    if (!(sectorcount)) sectorcount = 1; //fixes a bug.
    Peripheral_SingleByteWrite(CF_SECTOR_COUNT_REG, sectorcount);    // divide len into blocks
}

dwLBA += sectorcount;

// Execute the read command
if (verify)
    Peripheral_SingleByteWrite(CF_COMMAND_REG, CF_COMMAND_VERIFY_10);
else
    Peripheral_SingleByteWrite(CF_COMMAND_REG, CF_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 = getStatus(CF_STATUS_REG);
    }
    while(((driveStatus & (CF_STATUS_BUSY_BIT | CF_STATUS_DRQ_BIT)) != CF_STATUS_DRQ_BIT) && !(timeout-- & 0x8000)); // DO-WHILE!!!

    if (!(timeout & 0x8000) && !(driveStatus & CF_STATUS_ERROR_BIT))
    {
        // Normal case -- Got a sector. Send it to the host (in 8 chunks)
        for (i = 0; i < CF_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
                        //readPIO16(ATAPI_DATA_REG, IN2BUF, MAX_BULK_PACKET_SIZE);
                        Peripheral_AFIFOWordRead( CF_DATA_REG, MAX_BULK_PACKET_SIZE, IN2BUF);
    EPIO[IN2BUF_ID].bytes = MAX_BULK_PACKET_SIZE;
    }
    dataTransferLen -= CF_SECTOR_SIZE; // Returned a full sector.
}
else
// Timeout case -- Assume the host has caught up, send a short packet, then goto the stall code.
{
Peripheral_SingleByteRead( CF_ERROR_REG, chkbyte );
                        //readPIO8(ATAPI_ERROR_REG);

    IDEnop();
    failedIn();
    return(USBS_PASSED);
}
}
}
return(USBS_PASSED);
}

bit ideWriteCommand()
{
    BYTE savebc;
    BYTE driveStatus;
    WORD sectorcount;
    int timeout;
    BYTE i;
    DWORD dwLBA;
    //bit chk;

    Peripheral_SingleByteWrite(CF_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];
Peripheral_SingleByteWrite(CF_DRIVESEL_REG, 0xe0);
Peripheral_SingleByteWrite(CF_LBA_LSB_REG, ((char *) &dwLBA)[3]);
Peripheral_SingleByteWrite(CF_LBA_2SB_REG, ((char *) &dwLBA)[2]);
Peripheral_SingleByteWrite(CF_LBA_MSB_REG, ((char *) &dwLBA)[1]);
Peripheral_SingleByteWrite(CF_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
    Peripheral_SingleByteWrite(CF_DRIVESEL_REG, 0xe0);
    Peripheral_SingleByteWrite(CF_LBA_LSB_REG, ((char *) &dwLBA)[3]);
    Peripheral_SingleByteWrite(CF_LBA_2SB_REG, ((char *) &dwLBA)[2]);
    Peripheral_SingleByteWrite(CF_LBA_MSB_REG, ((char *) &dwLBA)[1]);
    Peripheral_SingleByteWrite(CF_DRIVESEL_REG, ((char *) &dwLBA)[0] | 0xe0);

    // First stuff the length register (number of sectors to write)
    if (dataTransferLenMSW & 0xfffe)
    {
        Peripheral_SingleByteWrite(CF_SECTOR_COUNT_REG, 0); // 0 means 256 blocks of 512
        sectorcount = 0x100; //0x100 = 256 sectors
    }
    else
    {
        sectorcount = dataTransferLenLSW/CF_SECTOR_SIZE + (dataTransferLenMSW & 1) * 0x80;
        if (!(sectorcount)) sectorcount = 1; //fixes a bug.
        Peripheral_SingleByteWrite(CF_SECTOR_COUNT_REG, sectorcount); // divide len into blocks
    }

    dwLBA += sectorcount;

    // Execute the write command
    Peripheral_SingleByteWrite(CF_COMMAND_REG, CF_COMMAND_WRITE_10);
}

```

```

while (sectorcount--)
{
    timeout = PROCESS_CBW_TIMEOUT_RELOAD;

    do
    {
        //driveStatus = readPIO8(ATAPI_STATUS_REG);
        Peripheral_SingleByteRead( CF_STATUS_REG, driveStatus );
    }
    while(((driveStatus & (CF_STATUS_BUSY_BIT | CF_STATUS_DRQ_BIT)) != CF_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 < CF_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)
            {
                Peripheral_AFIFOWordWrite( CF_DATA_REG, savebc, OUT2BUF );

                //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
            {
                Peripheral_AFIFOWordWrite( CF_DATA_REG, MAX_BULK_PACKET_SIZE, OUT2BUF );

                //writePIO16(ATAPI_DATA_REG, OUT2BUF, MAX_BULK_PACKET_SIZE);
                EPIO[OUT2BUF_ID].bytes = 0x40; // Give up the buffer
            }
        }
        dataTransferLen -= CF_SECTOR_SIZE; // Returned a full sector.
    }
    else
    {
        EZUSB_STALL_EP(OUT2BUF_ID);
    }
}

```

```

        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;
    //bit chk;
    BYTE chkbyte;
    // Pound the reset line
    //hardwareReset();
    // Wait for the completion
    for (count = 0, driveStatus = getStatus(CF_STATUS_REG);
        count < 100 && (driveStatus & CF_STATUS_BUSY_BIT);
        driveStatus = getStatus(CF_STATUS_REG),count++)
    {
        EZUSB_Delay(50);    // Wait 50 ms to be polite
    }
    Peripheral_SingleByteWrite(CF_DRIVESEL_REG, 0xe0);
    //writePIO8(ATA_DRIVESEL_REG, 0xe0);
    Peripheral_SingleByteRead( CF_LBA_LSB_REG, chkbyte );
    Peripheral_SingleByteRead( CF_LBA_2SB_REG, chkbyte );
    Peripheral_SingleByteRead( CF_LBA_MSB_REG, chkbyte );
    Peripheral_SingleByteRead( CF_DRIVESEL_REG, chkbyte );
    //readPIO8(ATA_LBA_LSB_REG);
    //readPIO8(ATA_LBA_2SB_REG);
    //readPIO8(ATA_LBA_MSB_REG);
    //readPIO8(ATA_DRIVESEL_REG);
}
BYTE getStatus(BYTE gaddr)
{
    BYTE result;
    //bit chk;
    Peripheral_SingleByteRead(gaddr, result);
    return(result);
}

```

```

}

bit checkForMedia()
{
    //check if there is media (check cfreset).
    //if cfreset is low, there is media.
    //check if the media has changed. If it has, reset the card.
    OEA = OEA & 0xFE;
    if (!(PINS_A & 0x01))
    {
    if (!prevMediaStatus)
        {
        sensePtr = senseMediaChanged;
        prevMediaStatus = 1;
        hardwareReset(); // reset the new card
                        //ATAPIIdDevice(); // Get the device name and serial number
        }
    return(1);
    }
    else
    {
    prevMediaStatus = 0;
    sensePtr = senseNoMedia;
    return(0);
    }
}

void initCF()
{
    prevMediaStatus = 0;
    sensePtr = senseOk;
    //flash = 1;
    //SCSIInquiryData_[1] = 0x80; // Set the RMB -- We're ALWAYS removable

    // Do the hardware setup for the port pins
    IFCONFIG = IFCONFIG & 0xF7;
    PORTACFG = 0x00;
    OEA = OEA | 0xFF;
    OUTA = 0x00; // Reset line high

    // Make Interrupt 0 level triggered (edge is the wrong polarity)
    //IT0 = 0;

```

```

// Set up the initial values
//OUTC = PORTC_IDLE_VALUE;

// Allow the values out into the world
//OEC = PORTC_OE;

FASTXFR = bmFBLK | bmRMOD1 | bmWMOD1;      //6,4,1

USBBAV = bmAVEN;      // disable the breakpoint line, enable autovector
}

/*void ATAPIIdDevice()
{
    BYTE driveStatus;

    {
        // Select device
        Peripheral_SingleByteWrite(CF_DRIVESEL_REG, 0xe0);

        if (waitForBusyBit(PROCESS_CBW_TIMEOUT_RELOAD) == USBS_FAILED)
            while (1)
                ;      // Unknown error condition -- What do we do?

        // Send Identify device command
        Peripheral_SingleByteWrite(CF_COMMAND_REG, IDE_COMMAND_ID_DEVICE);

        //waitForIntrq();

        // Wait for the register block to be non-busy and to have data ready
        do
        {
            driveStatus = getStatus(CF_STATUS_REG);
        }
        while((driveStatus & (CF_STATUS_BUSY_BIT | CF_STATUS_DRQ_BIT)) != (CF_STATUS_DRQ_BIT)); // Do-while

        if (driveStatus & CF_STATUS_ERROR_BIT)
            return;

        // Read the data from the drive
        readPIO16toXdata(CF_DATA_REG, halfKBuffer, BUFFER_SIZE); //BUFFER_SIZE

        // BUG -- what is this doing? Reduce the command set for ATA
        //if (!scsi)

```

```

// SCSIInquiryData_[SCSI_INQUIRY_DATA_FORMAT] = 0;

// This is actually smaller than a loop of 4!!
((BYTE *)&driveCapacity)[1] = halfKBuffer[3+IDE_ID_TOTAL_SECTORS_MSW];
((BYTE *)&driveCapacity)[0] = halfKBuffer[2+IDE_ID_TOTAL_SECTORS_MSW];
((BYTE *)&driveCapacity)[3] = halfKBuffer[1+IDE_ID_TOTAL_SECTORS_MSW];
((BYTE *)&driveCapacity)[2] = halfKBuffer[0+IDE_ID_TOTAL_SECTORS_MSW];
driveCapacity -= 1; // The command that reads drive capacity actually wants the last valid LBA.
return;
}
}

// Read data from the drive
// Issues repeated calls to readPIO16 to pull in multiple blocks
// of data from the drive
// Returns amount of data reported by drive
void readPIO16toXdata(char addr, char xdata *inbuffer, WORD count)
{
    WORD driveDataLen = 0;
    WORD saveDriveDataLen;
    BYTE driveStatus;
    WORD readLen;
    WORD timeout = PROCESS_CBW_TIMEOUT_RELOAD;

    for (driveStatus = 0; !(driveStatus & CF_STATUS_DRQ_BIT) && timeout-- > 0; )
        driveStatus = getStatus(CF_STATUS_REG);
    if (!timeout)
        return;

    count = saveDriveDataLen = CF_SECTOR_SIZE;
    driveDataLen = 0;

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

```

```

void hardwareReset()
{
    BYTE count;
    BYTE driveStatus;

    OEA = OEA | 0x01;
    OUTA = OUTA | 0x01;           // Reset line high
    EZUSB_Delay(200);           // Wait a while (minimum is 25us).
    OEA = OEA & 0xFE;           // Release the reset line. The CFCARD should pull reset low again.

    // Wait for the completion up to 5 seconds, then give up.
    for (count = 0, driveStatus = getStatus(CF_STATUS_REG);
        count < 10 && (driveStatus & CF_STATUS_BUSY_BIT);
        driveStatus = getStatus(CF_STATUS_REG),count++)
    {
        EZUSB_Delay(500);       // Wait 500 ms to be polite
    }
}

```

---

#### gpif1.c:

```

//
// This program configures the General Purpose Interface.
// Parts of this program are automatically generated using the GPIF Tool.
// Please do not modify sections of text which are marked as "DO NOT EDIT ...".
// You can modify the comments section of this GPIF program file using the dropdown menus
// and pop-up dialogs. These controls are available as hot spots in the text. Modifying the
// comments section will generate program code which will implement your GPIF program.
//
// DO NOT EDIT ...
// GPIF Initialization
// Interface Timing    Sync
// Data Bus Width     16 BIT
// Internal Ready Init IntRdy=1
// CTL Out Tristate-able Binary
// SingleWrite WF Select  3
// SingleRead WF Select  2
// FifoWrite WF Select   1
// FifoRead WF Select    0
// Data Bus Idle Drive  Tristate
// END DO NOT EDIT

```

```

// DO NOT EDIT ...
// GPIF Wave Names
// Wave 0 = Wave 0
// Wave 1 = Wave 1
// Wave 2 = Wave 2
// Wave 3 = Wave 3

// GPIF Ctrl Outputs Level
// CTL 0 = CTL 0 CMOS
// CTL 1 = CTL 1 CMOS
// CTL 2 = CTL 2 CMOS
// CTL 3 = CTL 3 CMOS
// CTL 4 = CTL 4 CMOS
// CTL 5 = CTL 5 CMOS

// GPIF Rdy Inputs
// RDY0 = RDY0
// RDY1 = RDY1
// RDY2 = RDY2
// RDY3 = RDY3
// RDY4 = RDY4
// RDY5 = RDY5
// FIFOflag = FIFOflag
// IntReady = IntReady
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 0: Wave 0
//
// Interval  0    1    2    3    4    5    6  Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData NextData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 2 Wait 2 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec

```

```

// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 0 0 1 1 1 1 1
// CTL 1 1 1 1 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 1: Wave 1
//
// Interval 0 1 2 3 4 5 6 Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData NextData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 7 Wait 3 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 1 1 1 1 1 1 1
// CTL 1 1 0 0 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 2: Wave 2
//
// Interval 0 1 2 3 4 5 6 Idle (7)
// _____
//

```

```

// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 2 Wait 2 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 0 0 1 1 1 1 1
// CTL 1 1 1 1 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1
// CTL 4 1 1 1 1 1 1 1 1
// CTL 5 1 1 1 1 1 1 1 1
//
// END DO NOT EDIT
// DO NOT EDIT ...
//
// GPIF Waveform 3: Wave 3
//
// Interval 0 1 2 3 4 5 6 Idle (7)
// _____
//
// AddrMode Same Val Same Val Same Val Same Val Same Val Same Val Same Val
// DataMode NO Data NO Data Activate Activate NO Data NO Data NO Data
// NextData SameData SameData SameData SameData SameData SameData SameData
// Int Trig No Int No Int No Int No Int No Int No Int No Int
// IF/Wait Wait 1 Wait 2 Wait 7 Wait 3 Wait 1 Wait 1 Wait 1
// Term A
// LFunc
// Term B
// Branch1
// Branch0
// Re-Exec
// Sngl/CRC Default Default Default Default Default Default Default
// CTL 0 1 1 1 1 1 1 1 1
// CTL 1 1 0 0 1 1 1 1 1
// CTL 2 1 1 1 1 1 1 1 1
// CTL 3 1 1 1 1 1 1 1 1

```

```

// CTL 4    1    1    1    1    1    1    1    1
// CTL 5    1    1    1    1    1    1    1    1
//
// END DO NOT EDIT

// GPIF Program Code

// DO NOT EDIT ...
#include "ezusb.h"
#include "ezregs.h"
#include "Fx.h"
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata WaveData[128] =
{
// Wave 0
/* LenBr */ 0x01, 0x02, 0x02, 0x02, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x04, 0x00,
/* Output*/ 0xFF, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun  */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 1
/* LenBr */ 0x01, 0x02, 0x07, 0x03, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x04, 0x00,
/* Output*/ 0xFF, 0xFD, 0xFD, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun  */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 2
/* LenBr */ 0x01, 0x02, 0x02, 0x02, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFE, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun  */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F,
// Wave 3
/* LenBr */ 0x01, 0x02, 0x07, 0x03, 0x01, 0x01, 0x01, 0x07,
/* Opcode*/ 0x00, 0x00, 0x02, 0x02, 0x00, 0x00, 0x00, 0x00,
/* Output*/ 0xFF, 0xFD, 0xFD, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
/* LFun  */ 0x00, 0x09, 0x12, 0x1B, 0x24, 0x2D, 0x36, 0x3F
};
// END DO NOT EDIT

// DO NOT EDIT ...
const char xdata InitData[7] =
{
/* Regs  */ 0xC0,0x00,0x00,0xFF,0x06,0xE4,0x11

```

```

};
// END DO NOT EDIT

// DO NOT EDIT ...
GpifInit(void)
{
    unsigned char xdata *Source;
    unsigned char xdata *Dest;
    unsigned int x;

    ABORT = 0;                // abort any pending operation
    READY = InitData[0];
    CTLOUTCFG = InitData[1];
    IDLE_CS = InitData[2];
    IDLE_CTLOUT = InitData[3];
    IFCONFIG = InitData[4];
    WFSELECT = InitData[5];
    ABSETUP |= InitData[6];

    Source = WaveData;       // Transfer the GPIF Tool generated data
    Dest = &WFDESC[0];

    for (x = 0; x < 128; x++)
        *Dest++ = *Source++;

    INT4SETUP = INT4SFC | INT4_INTERNAL; // setup INT4 as internal source
//    GENIE = 0x01; // Enable GPIF interrupt
    EIEX4 = 0x01;
    PORTSETUP |= 0x01;

    //PORTCCFG = 0xE0;
    PORTACFG = 0x02;
    OEA = 0xFD;
//    OUTA = 0x03;        //not reset, /CE1-low, /CE2-low
//    EZUSB_Delay(1);
    OUTA = 0x02;
}
// END DO NOT EDIT

// TO DO: You may add additional code below.
//

#define TMOUT 0x0020        // Default Timeout TODO: Set this appropriately

```

```

// write byte to PERIPHERAL, using GPIF
bit Peripheral_SingleByteWrite( BYTE gaddr, BYTE gdata )
{
    unsigned char transaction_err = 0x00;

    //OEC = 0x1F;
    //OUTC = gaddr & 0x0F;

    OUTA = 0x0A | (gaddr*16);

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    SGLDATLTRIG = gdata;        // initiate GPIF write transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );           // error has occurred
        }
    }
    return( 1 );
}

```

```

// read byte from PERIPHERAL, using GPIF
bit Peripheral_SingleByteRead( BYTE gaddr, BYTE xdata *gdata )
{
    unsigned char g_data = 0x00;
    unsigned char transaction_err = 0x00;

    OUTA = 0x0A | (gaddr*16);

    //GPIFADRL = gaddr;          // setup GPIF address ADR0-ADR5
    g_data = SGLDATLTRIG;        // initiate GPIF read transaction

    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;

```

```

    return( 0 );      // error has occurred
}
}

*gdata = SGLDATLNTRIG;

return( 1 );
}

// write word(s) to PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf )
{
    unsigned char transaction_err = 0x00;

        OUTA = 0x02 | (gaddr*16);

//GPIFADRL = gaddr;    // setup GPIF address ADR0-ADR5
DMASRC = outbuf;      // Typically points to endp buffer
DMADEST = &AOUTDATA; // point to FIFO-A
DMALEN = xfrcnt;
EA = 0;               // protect DMA from interrupts occurring in Block0
DMAGO = xfrcnt;
AOUTTC = xfrcnt >> 1; // divide by 2 for 16 bit transactions
ATRIG = xfrcnt;      // write to ATRIG initiates
                    // FIFO -> GPIF transaction(s)
while( !( DMAGO & 0x80 ) )
{
    ; // wait here for the DMA to complete
}
EA = 1;               // Enable interrupts...

while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 Done bit
{
    // transaction completed
    if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
    {
        ABORT = 0x01;
        return( 0 );      // error has occurred
    }
}

return( 1 );

```

```

}

// read word(s) from PERIPHERAL, using GPIF, DMA, and slave FIFOA
bit Peripheral_AFIFOWordRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf)
{
    BYTE transaction_err = 0x00;
    BYTE gxfr = 0x00;

    OUTA = 0x02 | (gaddr*16);

    //GPIFADRL = gaddr;           // setup GPIF address ADR0-ADR5
    AINTC = xfrcnt >> 1;         // divide by 2 for 16 bit interface
    gxfr = ATRIG;                // read from ATRIG initiates
                                // GPIF -> FIFO transaction(s)
    while( !( IDLE_CS & 0x80 ) ) // poll IDLE_CS.7 GPIF Done bit
    {
        if( ++transaction_err > TMOUT ) // trap GPIF transaction for TMOUT
        {
            ABORT = 0x01;
            return( 0 );           // an error has occurred
        }
    }

    DMASRC = &AINDATA;           // point to FIFO-A
    DMADEST = inbuf;             // Typically points to endp buffer
    DMALEN = xfrcnt;

    EA = 0;                      // protect DMA from interrupts occurring in Block0
    DMAGO = xfrcnt;              // writing any value to DMAGO starts the DMA

    while( !( DMAGO & 0x80 ) )
    {
        ; // wait here for the DMA to complete
    }
    EA = 1;                       // Enable interrupts...

    return( 1 );
}

```

---

gpif.h:

```

extern bit Peripheral_SingleByteRead( BYTE gaddr, BYTE xdata *gdata );
extern void GpifInit(void);
extern bit Peripheral_SingleByteWrite( BYTE gaddr, BYTE gdata );
extern bit Peripheral_AFIFOWordWrite( BYTE gaddr, BYTE xfrcnt, BYTE xdata *outbuf );
extern bit Peripheral_AFIFOWordRead( BYTE gaddr, BYTE xfrcnt, BYTE xdata *inbuf );

```

---

## Cfconst.h:

```

#define dataTransferLenMSW      ((WORD *) (&dataTransferLen))[0]
#define dataTransferLenLSW      ((WORD *) (&dataTransferLen))[1]
#define dataTransferLenMSB      ((BYTE *) (&dataTransferLen))[0]
#define dataTransferLen2SB      ((BYTE *) (&dataTransferLen))[1]
#define dataTransferLen3SB      ((BYTE *) (&dataTransferLen))[2]
#define dataTransferLenLSB      ((BYTE *) (&dataTransferLen))[3]

#define min(a,b) (((a)<(b))?(a):(b))

// Local defines from the mass storage class spec
#define CBW_TAG                  4
#define CBW_DATA_TRANSFER_LEN_LSB  8
#define CBW_DATA_TRANSFER_LEN_MSB  9
#define CBW_FLAGS                12
#define CBW_FLAGS_DIR_BIT        0x80
#define CBW_LUN                  13
#define CBW_CBW_LEN              14
#define CBW_CBW_LEN_MASK        0x1f
#define CBW_DATA_START           15

#define IDE_ID_TOTAL_SECTORS_MSW  60*2
#define IDE_COMMAND_ID_DEVICE    0xec
#define BUFFER_SIZE 0x200

#define SCSIInquiryData_ ((BYTE xdata *) SCSIInquiryData)
//RBC command
#define TEST_UNIT_READY 0x00
#define REQUEST_SENSE 0x03
#define INQUIRY 0x12
#define MODE_SELECT_06 0x15
#define READ_10 0x28
#define MODE_SENSE_06 0x1A

```

```

#define STOP_START_UNIT      0x1B
#define PREVENT_ALLOW_MEDIUM_REMOVAL      0x1E
#define READ_CAPACITY      0x25
#define WRITE_10      0x2A
#define VERIFY_10      0x2F

//CF registers
#define CF_DATA_REG      0x00
#define CF_ERROR_REG      0x01
#define CF_SECTOR_COUNT_REG      0x02
#define CF_LBA_LSB_REG      0x03
#define CF_LBA_2SB_REG      0x04
#define CF_LBA_MSB_REG      0x05
#define CF_DRIVESEL_REG      0x06
#define CF_STATUS_REG      0x07
#define CF_COMMAND_REG      0x07

//CF command
#define CF_COMMAND_READ_10      0x20
#define CF_COMMAND_WRITE_10      0x30
#define CF_COMMAND_VERIFY_10      0x40

//CF variables
#define CF_SECTOR_SIZE      0x200 //512 bytes
#define MAX_BULK_PACKET_SIZE      0x40 //64 bytes max bulk size
#define PROCESS_CBW_TIMEOUT_RELOAD      0x7000

#define USBS_FAILED      0
#define USBS_PASSED      1
#define CF_STATUS_BUSY_BIT      0x80
#define CF_STATUS_DRQ_BIT      0x08
#define CF_STATUS_ERROR_BIT      0x01

extern code BYTE SCSIInquiryData[44];

extern DWORD dataTransferLen;
extern DWORD driveCapacity;

extern xdata BYTE halfKBuffer[BUFFER_SIZE];

```

---

periph.h:

```
extern void mymemmovexx(BYTE xdata * dest, BYTE xdata * src, BYTE len);  
extern void sendUSBS(BYTE passOrFail);  
extern void failedIn();  
extern bit waitForBusyBit(WORD timeout);
```

---

## Bugs in the Hardware Design

There was only 1 bug in the hardware design (atleast only one that we have found so far). The pads for the Cypress 128-pin EZ-USB FX chip were too short. This made soldering the chip to the board difficult. However we have corrected the error (increased the length by 20 mils) and updated the footprints.

## Possible Changes in the Design

- ✿ In this board, as in the first one, we don't have a button to perform an electrical disconnect of the board from the USB port. A disconnect button would be a very useful feature to have especially during code development (when you have to disconnect and reconnect each time a fresh piece of code is loaded).
- ✿ Another change we could make to the board is, to use power and ground fills instead of just traces around the board. This would reduce the ground bounce and give us much cleaner supplies.

## References

### **PCB board**

[www.4pcb.com](http://www.4pcb.com)

### **Free samples**

Voltage regulator, RS-232 Transceiver: [www.maxim-ic.com](http://www.maxim-ic.com)

Connectors, jumpers, pins: [www.cranecconnectors.com](http://www.cranecconnectors.com)

USB port Transient Suppressor: [www.ti.com](http://www.ti.com)

Compact Flash adaptor: [www.samtec.com](http://www.samtec.com)

Cypress Fx chip: [www.cypress.com](http://www.cypress.com)

IO expander: [www.philips.com](http://www.philips.com)

EEPROM: [www.fairchild.com](http://www.fairchild.com)

LED: [www.lumex.com](http://www.lumex.com)

### **Useful websites**

Compact flash: [www.sandisk.com](http://www.sandisk.com)

Reduced block commands: [http://www.ncits.org/standards/list\\_NCITS.htm](http://www.ncits.org/standards/list_NCITS.htm)

USB mass storage class (bulk-only transfer):

[http://www.usb.org/developers/devclass\\_docs/usbmassbulk\\_10.pdf](http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf)

USB mass storage class specification overview:

[http://www.usb.org/developers/devclass\\_docs/usbmassover\\_11.pdf](http://www.usb.org/developers/devclass_docs/usbmassover_11.pdf)

USB Device class definition for HID:

[http://www.usb.org/developers/devclass\\_docs/HID1\\_11.pdf](http://www.usb.org/developers/devclass_docs/HID1_11.pdf)

USB specification and documentation: [www.usb.org](http://www.usb.org)

Useful source codes, reference designs: [www.cypress.com](http://www.cypress.com)

### **Useful information in development board CD**

Quick training and examples: <C:\Cypress\USB\Application Reference Materials>About USB\USB Interfacing Training Series\Cypress USB Course>

Hex2bix program: <C:\Cypress\USB\Bin>

Basic Firmware structure: <C:\Cypress\USB\Target\Fw\Ezusb>

Schematics, Layout for Cypress Development Boards: <C:\Cypress\USB\Hardware\EZ-USB Fx>

Basic Firmware explanation: <C:\Cypress\USB\Doc\EZ-USB General>

Control panel explanation: <C:\Cypress\USB\Doc\EZ-USB General>