Using PortA on AN2131-DK001 “Cypress USB Full-Speed Development Kit”

Introduction

This paper discusses how to transfer data over Endpoint 2 then out EZ-USB PORTA using EZ-USB Control Panel.

We’ll start on the basis that you have already received your development kit and installed the software that came on the CDROM. Hopefully you didn’t miss out on the tutorial under EZ-USB Control Panel **Help | Contents and Tutorial** menu.

Background

EZ-USB does not implement port IO pins in the same manner as a standard 8051. The standard 8051 maps these ports in SFR space, allowing port pins to be manipulated via single cycle instructions (**mov**, **setb**, **clr**, etc.). EZ-USB maps these ports in XDATA space. Typically, what this means is that in order to manipulate port pins with EZ-USB you must use the **movx** instruction.

Initialization

We’ll start with a frameworks-based variant of the EP_PAIR example. The simplest way to accomplish this is to copy the entire EP_PAIR folder to a temp directory and rename it as porta then move it back under the examples directory. At this point you can rename *ep_pair.c* to *porta.c* and modify the *build utility (*.bat, *.prj, or *.uv2)* file appropriately.

EZ-USB PORTA defaults as IO and three-stated (input direction) during a power-on reset. For our implementation we want these pins configured for output. Below is the “C” code for EZ-USB PORTA initialization in *TD_Init()*:

```
PORTACFG = 0x00; // use all PORTA pins as IO
OEA = 0xFF; // and then setup all as output
```

The register PORTACFG contains all the bits necessary to configure EZ-USB PORTA pins as either general purpose IO (set bit to 0) or as an alternate function (set bit to 1), see TRM for details.

The register OEA contains all the bits to setup each EZ-USB PORTA pin for either input (set bit to 0) or output (set bit to 1) direction.

To enable the frameworks for data transfer using Endpoint 2 out we need to set the appropriate bit in the OUT07VAL register. Planning ahead we could also enable Endpoint 2 in using the IN07VAL register.

Once enabled we should arm the Endpoint by writing any value to its byte count register. This effectively sets the Endpoints busy bit making it available to the EZ-USB core.

Example Description

Now that EZ-USB PORTA is properly set up for our implementation and the Endpoint is ready to receive data from the host we can move on to our application specific implementation of the *TD_Poll()* routine in the firmware frameworks.

The EZ-USB core clears the Endpoint’s busy bit and writes to the byte count register when the host data is available in the OUT buffer. This mechanism makes for very straight forward and easy to read *TD_Poll()* firmware.

Since we only have one goal (to transfer USB data out EZ-USB PORTA) our firmware can sit in a tight loop looking for the busy bit to clear. Once our firmware detects data in OUT2BUF, by polling the busy bit, we set up a **for** loop to transfer these bytes out PORTA via writing to the OUTA register.

The final step is to re-arm the Endpoint to receive more data by writing to the byte count register.

Example Overview

Below in Listing 1 are our specific *TD_Init()* and *TD_Poll()* routines:

Listing 1

```
 1  void TD_Init(void)         // Called once at startup
 2  {
 3
 4    // PORTA setup for output
 5    PORTACFG = 0x00;         // use all porta pins a IO
 6    OEA = 0xFF;              // and then all as outputs
 7
 8      // porta application specific USB register initialization
 9    IN07VAL = bmEP2;         // using endpoint 2 in
10    OUT07VAL = bmEP2;        // using endpoint 2 out
11
12    // arm endpoint 2 out to receive data
13    EPIO[OUT2BUF_ID].bytes = 0x00;
14
15    // setup breakpoint to trigger each time TD_Poll() gets called
16    BPADDR = (WORD)TD_Poll;
17    USBBAV |= bmBPEN;        // enable breakpoint
18    USBBAV &= ~bmBPPULSE;
19
20    Rwuen = TRUE;               // Enable remote-wakeup
21
22  }
23
24  void TD_Poll(void)         // Called repeatedly while the device is idle
25  {
26    BYTE i = 0x00;
27
28    if( !( EPIO[ OUT2BUF_ID ].cntrl & bmEPBUSY ) )
29    { // there's data in OUT2BUF
30      for( i = 0x00; i < EPIO[ OUT2BUF_ID ].bytes; ++i )
31      {
32        OUTA = OUT2BUF[ i ];       // tx byte using porta
33      }
34      EPIO[ OUT2BUF_ID ].bytes = 0x00;
35    }
36
37    // toggle green breakpoint/monitor LED for visual notification...
38    EZUSB_Delay( 100 );  // to see the green LED ON
39    USBBAV |= bmBREAK;
40    EZUSB_Delay( 100 );  // to see the green LED OFF
41  }
```

After modifying *porta.c* for the above two functions and rebuilding the project files we are ready for testing.

Download *porta.hex* to the target using EZ-USB Control Panel. Notice that the green monitor LED is flashing. Then set up BulkTrans for Endpoint 2 out and HexData for AA 55 AA 55 AA 55 AA 55 AA 55. To initiate the transfer click the “BulkTrans” button.

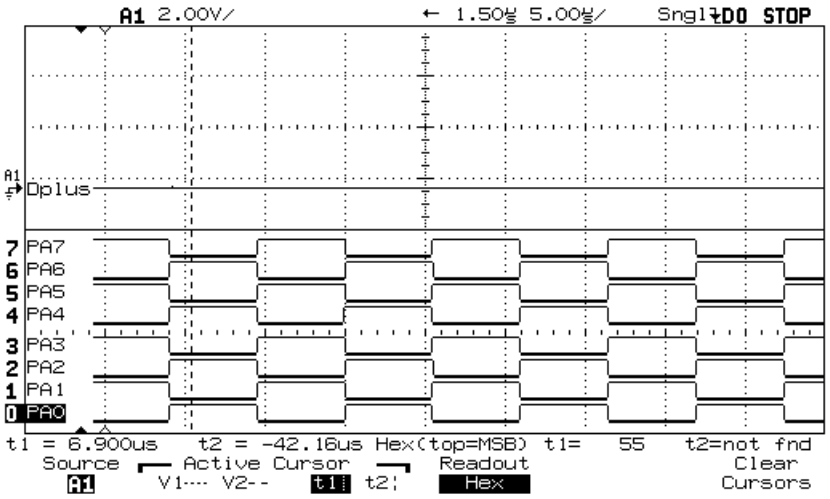

Figure 1.

Figure 1 shows a logic analyzer screen capture of EZ-USB PORTA’s output after transferring alternating AA and 55 via EZ-USB Control Panel.

It may be important to note that with this implementation the maximum data throughput that can be transmitted every SOF is approximately 128 bytes. This is due to the latency of re-arming the Endpoint and making it available to the host. Examining the assembler created by the C compiler in the listing file will explain why this is happening. Worst case it takes 2240 instruction cycles (~373 µs) for every 64 byte packet. Note that this latency is implementation specific. You could ‘un-roll” the *for* loop in *TD_Poll()* to improve the bandwidth performance.

Conclusion

EZ-USB implements port IO differently than a standard 8051. The main difference being that EZ-USB maps these ports in XDATA space.