AN1215

Basic Embedded Host Using the SL811HST

Author: Narayana Murthy Mandava
Associated Project: No
Associated Part Family: SL811HST-AXC
Software Version: None
Related Application Notes: AN1137

AN1215 explains the SL811HST functionality as a USB Full/Low-Speed Host. The document begins with a description of signal interconnects between an external processor and SL811HST. It then introduces the SL811 register sets and their functionality. The host mode firmware example code, which comes along with the SL811HST development kit (CY3662), is used as a reference. This note provides a background on the SL811 host mode functionality to help you get started with host mode designs.

1 Introduction

The SL811HST is a full-featured USB embedded host controller. It uses a standard address/data bus typical of most 16- and 32-bit embedded processors as well as some 8-bit microcontrollers. This application note addresses the usage of the SL811HST in an embedded USB host application.

1.1 System Interface

The SL811HST incorporates an industry-standard address/data bus. The embedded processor signals have the following requirements:

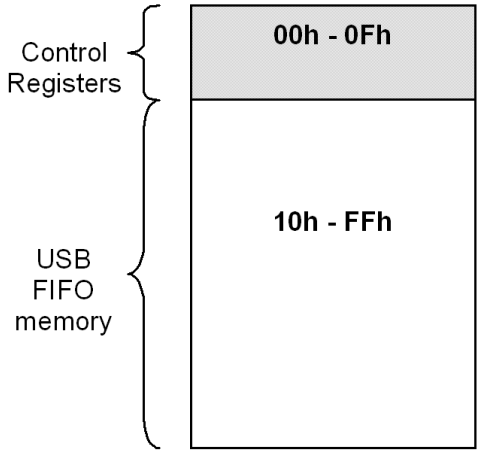
- Active LOW CHIP SELECT signal
- Active LOW READ signal
- Active LOW WRITE signal
- Active HIGH INTERRUPT signal
- Address bus or GPIO
- Data bus, at least 8 bits wide
- GPIO to drive RESET and USB bus power enable

See the application note Interfacing an External Processor to the SL811HS for more details on host circuitry configuration. This application note provides complete details and examples of the signaling interface to the SL811HST. Most microcontrollers will be able to interface to the SL811HST with little or no glue logic.

2 Programming Interface

The SL811HST uses a memory mapped interface with an 8-bit address range. The SL811HST supports both a host and peripheral interface; however, only the host registers are described in this document. The first 16 addresses (00h-0Fh) are filled with 20 registers used to control the USB host SIE. The addresses ranging from 10h-FFh are used as user assignable USB FIFO buffers. Figure 1 shows the SL811HST memory map.

Figure 1. SL811HST Memory Map



The 20 host control registers are used to enable transactions, interrupts, and report status. Table 1 is a summary of the SL811HST register set. The table is followed by a brief description of each register.

Table 1. SL811HST Register Set Summary

Addr.	Write Function	Read Function
0x00	USB-A Control	USB-A Control
0x01	USB-A Address	USB-A Address
0x02	USB-A Length	USB-A Length
0x03	USB-A PID/EP	USB-A Status
0x04	USB-A Address	USB-A Count
0x05	Ctrl1	Ctrl1
0x06	Int. Enable	Int. Enable
0x08	USB-B Control	USB-B Control
0x09	USB-B Address	USB-B Address
0x0A	USB-B Length	USB-B Length
0x0B	USB-B PID/EP	USB-B Status
0x0C	USB-B Address	USB-B Count
0x0D	Int. Status	Int. Status
0x0E	SOF Low	HW Revision
0x0F	SOF High/Ctrl2	SOF High/Ctrl2

USB-A/B Host Control (0x00, 0x08, R/W) – This register is used to provide control over basic host transactions. For example, the register enables USB transactions, sets the transaction direction, and controls the data toggle.

USB-A/B Base Address (0x01, 0x09, R/W) – This register acts as a memory pointer in the range of 10h-FFh. Data that is sent to a USB peripheral is gathered from this internal memory location and sent over the USB. Data reported from a USB peripheral is put at the memory location pointed to by this register.

USB-A/B Base Length (0x02, 0x0A, R/W) – The base length is used to determine the maximum length of a transaction. When the SL811HST sends data to a USB peripheral this register determines the length of the data in the packet. When the USB peripheral reports data back to the host this register determines the maximum data length that will be accepted.

USB-A/B PID/Endpoint (0x03, 0x0B, W) – This register contains the host PID (i.e., SETUP, IN, OUT) and the target endpoint number.

USB-A/B Status (0x03, 0x0B, R) – This register contains the status of the last performed USB transaction. The status includes the received PID (ACK, NAK, and STALL), data toggle, and any error condition.

USB-A/B Address (0x04, 0x0C, W) – This register contains the USB peripheral device address

USB-A/B Transfer Count (0x04, 0x0C, R) – This register contains the residual transfer count after a USB transaction has taken place. In either transfer direction this register value represents the difference between the value written to the Base Length register and the actual number of bytes written from/read into the SL811HST internal memory. If the peripheral tries to send too large of a packet for the SL811HST to handle, the error will be noted in the Status register.

Control 1 (0x05, R/W) – This register enables SOF generation, resets the SIE, allows software to set the USB data line states, sets the USB bus speed, and suspends the SL811HST. The ability to set the USB data lines states is particularly useful for signaling a USB bus reset.

Interrupt Enable (0x06, R/W) – This register allows software to enable an interrupt signal (INTRQ HIGH) on certain events. These events include transaction completion, SOF, device insertion/removal, and resume signaling detection.

Interrupt Status (0x0D, R/W) – This register is read by the external processor upon an interrupt event to find which event caused the interrupt. The events reported in this register correspond to the events enabled in the Interrupt Enable register. The interrupt is deasserted by writing a “1” to any asserted interrupt bit.

SOF Counter Low (0x0E, W) – Sets the low byte of the timer that tracks SOF timing. This register should be written with E0h after reset.

Hardware Revision (0x0E, R) – This register allows device firmware to read the current silicon revision. See the SL81HST data sheet for the most current valid values.

SOF Counter High/Control 2 (0x0F, R/W) – Sets the high byte of the timer that tracks SOF timing, allows software to swap D±, and selects host or peripheral modes. This register should be written with the value AEh after reset to enable host mode and proper SOF timing. The SOF Counter High/Low registers must be initialized before enabling SOF generation.

Two transaction engines, USB-A and USB-B, are provided so that one transaction can be set up while the other is taking place. The transaction engines are symmetric, so it does not matter which one is used and device software is not required to interact with both engines. In fact some simpler applications, such as using a mouse and keyboard on a set top box, would typically only use one engine because throughput requirements for these devices are minuscule and software complexity can be reduced by dealing with one engine only. Both transaction engines are more commonly used in high-throughput applications such as video or mass storage.

3 USB Host Operation

The USB host functionality of SL811HST is demonstrated using the **Emb_host** firmware example as a reference in this application note. The firmware is part of the CY3662 DVK software developed with the hardware set up available along with the DVK kit. The kit basically contains a SL811HST daughter card memory mapped to an 8051-based board. The following figure shows the CY3662 kit hardware.

Figure 2. CY3662 Development Kit

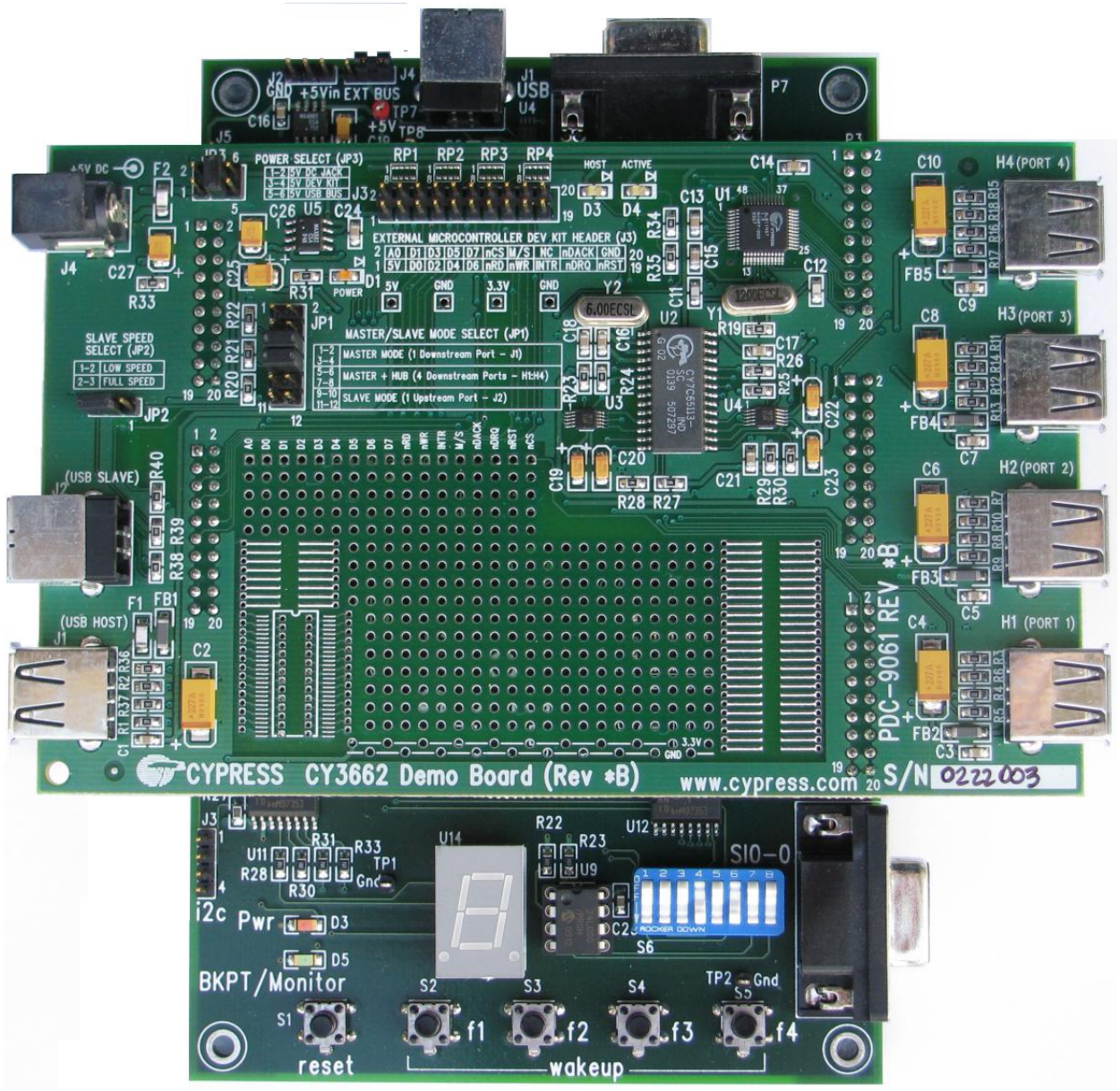


Figure 2 shows the SL811HST daughter card supporting both Slave and Host mode ports. Additionally, it includes an external full-speed hub (CY7C65113) to enumerate devices on four downstream ports. The **Emb_Host** firmware uses the combined (HUB + Host) functionality to enumerate as a HID mouse when connected to any of the four downstream ports of the HUB. The following list summarizes the source files in the firmware project and list of tasks handled inside them. This application note is intended only for the host functionality of SL811HST.

***fw.c*:** Handles enumeration of the EZ-USB when connected to the PCs or Laptop

***host2131.c*:** Handles the interface to the EZ-USB board. This includes sending bulk command data to the 8051 for the SL811HST and returning data captured from the slave device back to the PC. Basically, it handles the IN and OUT transfers to and from Host PC using EZ-USB.

***dscr.a51*:** This file contains descriptor information.

***hub_811.c*:** The SL811HST daughter card also integrates Cypress's full-speed hub (CY7C65113) to detect slave devices on four downstream ports. After detecting the device attachment/detachment speed, it finally calls host routines to enumerate the slave device

***host_811.c*, *host_811.h*:** The source file *host_811.c* contains the source code for SL811HST to perform as a host. The host is the initiator of any USB bus events when a slave device is connected at the other end. The following description highlights the sequence of events that occur when a slave device is connected to SL811HST as a host. The SL811HST host mode operation can be classified into four stages:

- Basic SL811HST initialization
- USB slave detection
- Enumeration process
- USB data transfers

Basic SL811HST Initialization: There are three basic steps during Sl811 initialization:

- Setting I/O pin modes of PORT A, B, C to communicate with SL811HST. These I/O pins act as control and data bus signals to configure SL811 memory using EZ-USB. This configuration method is specific to the external processor to which SL811 is connected
- Hardware reset of SL811HST and eventually clearing reset after 5 msec
- Setting SL811HST to host mode through PORT B I/O pins. The following code highlights the steps in the host source code. The EZ-USB has an 8051 controller inside it and this external controller configures SL811HST registers and buffer memory through port I/O pins. Hence, the firmware can be compiled and edited using Keil uVision2 IDE.

```
      void sl811h_init(void)
{
.............
.....................
SLAVE_FOUND = FALSE;
SLAVE_ENUMERATED = FALSE;
      ..................
/*Step a.SL811H + EZUSB I/Os setup*/
PORTBCFG &= 0xAC;          //Select i/o function for PB6, PB4, PB1, PB0
OEB      |= 0x43;          //Set PB6(Output), PB4(Input), PB1(Output), PB0(Output)
OUTB     |= 0x43;          //Default output high
PORTACFG &= 0x0F;          //Select i/o function for PA7~PA4
OEA      |= 0xF0;          //Set PA7~PA4(Output)
OUTA     |= 0xF0;          //Default output high

/*Step b. SL811 Hard reset and clear */
OUTB &= ~nRESET;           //reset SL811HST
EZUSB_Delay(5);            // for 5ms
OUTB |= nRESET;            // clear reset

/*Step c. Setting SL811 Host mode */
OUTB &= ~nHOST_SLAVE_MODE; // set to Host mode
}
```

USB Slave Detection: There are several stages in SL811HST host firmware to detect a USB peripheral. Following are the list of steps in sequence to detect a slave**.**

a. Monitoring firmware events: The SL811HST Host firmware is designed in such a way to monitor some of the events like slave detection occurring across the firmware. The TD_poll() is repeatedly called inside main( ) function for this purpose. Following is the code snippet.

```
/*EZ-USB AN2131 code from host2131.c */
      void main()
{
          ..............
          ..............
```

```
        TD_init()
            while(TRUE)                                  // Main Loop{

            ............
            ..........

            TD_poll( )

        }
    }

    /* Continuous looping code defined in fw.c of EZ-USB AN2131*/
    void TD_poll()
    {
        .........
        slave_detect( )/*SL811 function to detect USB events */
        ..........
    }
```

b. Enumeration Completion: The Host firmware example in the CY3662 kit is designed in such a way that unless EZ-USB completes its enumeration process SL811HST will not detect USB bus events. Setting CONFIG_DONE as TRUE indicates set_config() request is processed during enumeration and SL811HST subsequently can detect the USB events. Inside the slave_detect() function CONFIG_DONE variable is monitored to check if enumeration is complete.

c. Detecting USB Bus events: After enumeration stage there are series of USB Bus events that SL811HST detects using speed_detect() function in the host firmware. The following is the brief list of USB bus events and firmware configurations inside speed_detect()

i. Detect USB reset

ii. Detect Full speed/Low speed device

iii. Setting up SOF counter and SOF generation.

All the above events from steps (b) ,(c) are shown in the following code snippets

```
int slave_detect(void)
{

/* Step a. Wait for EZUSB enumeration */
if(!CONFIG_DONE)
// start SL811H after EZ-USB is configured
       return 0;
           if(!SLAVE_ENUMERATED)
// only if slave is not configured
                speed_detect();
// wait for an USB device to be inserted to
            if(SLAVE_FOUND)
// the SL811HST host

            if(EnumUsbDev(1))
// enumerate USB device, assign USB address = #1
            {
            SLAVE_ENUMERATED = TRUE;   // Set slave USB device enumerated flag
            uHub.bPortPresent[1] = 1;  // set device addr #1 present
            Set_ezDEV(1);
// inform master of new attach/detach
            }                   else
          {
          if(Slave_Detach())
// test for slave device detach ???

          ..................
```

```
        }
                    int speed_detect()
                    {
SL811Write(cSOFcnt,0xAE);  // Set SOF high counter, no change D+/D-, host mode
                           SL811Write(CtrlReg,0x08);
// Reset USB engine, full-speed setup, suspend disable
                           EZUSB_Delay(10);
// Delay for HW stabilize
                           SL811Write(CtrlReg,0x00);
// Set to normal operation
                           SL811Write(IntEna,0x61);
// USB-A, Insert/Remove, USB_Resume.
                           SL811Write(IntStatus,INT_CLEAR);
// Clear Interrupt enable status
                           EZUSB_Delay(10);    // Delay for HW stabilize
              ...................
                           ..................
                    }
if(SL811Read(IntStatus)&USB_RESET)
    {                                      // test for USB reset
        SL811Write(IntStatus,INT_CLEAR);   // Clear Interrupt enable status
        EZUSB_Delay(30);                   // Blink LED - waiting for slave USB plug-in
        OUTB ^= ACTIVE_BLINK;              // Blink Active LED
        OUTA |= PORTX_LED;                 // clear debug LEDs
        return 0;                          // exit speed_detect()
    }

    if((SL811Read(IntStatus)&USB_DPLUS)==0)     // Checking full or low speed
    {                                      // ** Low Speed is detected ** //
        SL811Write(cSOFcnt,0xEE);          // Set up host and low speed direct and SOF
                                           cnt
        SL811Write(cDATASet,0xE0);         // SOF Counter Low = 0xE0; 1ms interval
        SL811Write(CtrlReg,0x21);          // Setup 6MHz and EOP enable
        uHub.bPortSpeed[1] = 1;            // low speed for Device #1
        FULL_SPEED = FALSE;                // low speed device flag
    }
    else
    {                                      // ** Full Speed is detected ** //
        SL811Write(cSOFcnt,0xAE);          // Set up host & full speed direct and SOF cnt
        SL811Write(cDATASet,0xE0);         // SOF Counter Low = 0xE0; 1ms interval
        SL811Write(CtrlReg,0x05);          // Setup 48MHz and SOF enable
        uHub.bPortSpeed[1] = 0;            // full speed for Device #1
    }

    OUTB |= ACTIVE_BLINK;                  // clear Active LED
    SLAVE_FOUND = TRUE;                    // Set USB device found flag
    SLAVE_ENUMERATED = FALSE;              / no slave device enumeration

    SL811Write(EP0Status,0x50);            // Setup SOF Token, EP0
    SL811Write(EP0Counter,0x00);
// reset to zero count
    SL811Write(EP0Control,0x01);           // start generate SOF or EOP

    EZUSB_Delay(25);                       // Hub required approx. 24.1mS
    SL811Write(IntStatus,INT_CLEAR);       // Clear Interrupt status
    return 0;                              // exit speed_detect();
```

USB Enumeration Requests: *EnumUsbDev()* routine supports entire USB enumeration of a slave device. Following are series of USB events and requests that SL811HST as Host initiates when enumerating the slave device:

a. USB bus reset – After a device is attached the host is required to generate a USB bus reset. USB reset through a hub is not discussed in this document, however more information can be found in the USB specification. USB bus reset is generated when the SL811HST drives both D+ and D– LOW for 50 ms or more. Bits 3 and 4 of the Control 1 register allow software to directly control the states of the SL811HST D± pins and set both pins LOW. After 50 ms, control of the D± pins should be returned to the SL811HST SIE via the same control bits. The code below shows the USB reset event assertion:

```
int EnumUsbDev(BYTE usbaddr)
{
    ........................
    ........................
    // Reset only Slave device attached directly
    uDev[0].wPayLoad[0] = 64;                   // default 64-byte payload of Endpoint
0, address #0
    if(usbaddr == 1)                            // bus reset for the device attached to
SL811HS only
       USBReset();                              // that will always have the USB
address = 0x01 (for a hub)
     EZUSB_Delay(25);


// UsbReset during enumeration of device attached directly to SL811HS
void USBReset()
{
    BYTE tmp;
     tmp =  SL811Read(CtrlReg);
    SL811Write(CtrlReg,tmp|0x08);
    EZUSB_Delay(25);
     SL811Write(CtrlReg,tmp);
}
```

b. USB Enumeration requests on EP0: SL811HST host firmware initiates enumeration requests in the following sequential order on control endpoint(EP0):

GET_DEVICE_DESC:On Enpoint 0 addr =0 this request is initiated first for VID/PID and other parameters as shown below:

```
    // Get USB Device Descriptors on EP0 & Addr 0
    // with default 64-byte payload
    //-----------------------------------------------
    pDev =(pDevDesc)DBUF;                       // ask for 64 bytes on Addr #0
    if (!GetDesc(uAddr,DEVICE,0,18,DBUF))       // and determine the wPayload size
       return FALSE;                            // get correct wPayload of Endpoint 0
    uDev[usbaddr].wPayLoad[0]=pDev->bMaxPacketSize0;// on current non-zero USB address


/*Function defintion*/
int GetDesc(BYTE usbaddr, WORD wValue,  WORD wIndex, WORD wLen, BYTE *desc)
{
    return VendorCmd(usbaddr, 0x80, GET_DESCRIPTOR, wValue, wIndex,
wLen, desc);
```

SET_ADDR: The SL811 Host sets the device addr as 1 as shown in the following code

```
//-----------------------------------------------
    // Set Slave USB Device Address
    //-----------------------------------------------
    if (!SetAddress(usbaddr))                   // set to specific USB address
       return FALSE;                            //
    uAddr = usbaddr;                            // transfer using this new address
```

```
int SetAddress(WORD addr)
{
    return VendorCmd(0,0,SET_ADDRESS, WordSwap(addr), 0, 0, NULL);}
```

GET_DEVICE_DESC: Using the fixed device addr(1) the Host again requests entire device descriptor using the following code:

```
    //-------------------------------------------------
    // Get USB Device Descriptors on EP0 & Addr X
    //-------------------------------------------------
    if (!GetDesc(uAddr,DEVICE,0,(pDev->bLength),DBUF))
        return FALSE;                                       // For this current
device:
    uDev[usbaddr].wVID      = pDev->idVendor;               // save VID
    uDev[usbaddr].wPID      = pDev->idProduct;              // save PID
    uDev[usbaddr].iMfg      = pDev->iManufacturer;          // save Mfg Index
    uDev[usbaddr].iPdt      = pDev->iProduct;               // save Product Index

int GetDesc(BYTE usbaddr, WORD wValue,  WORD wIndex, WORD wLen, BYTE *desc)
{
    return VendorCmd(usbaddr, 0x80, GET_DESCRIPTOR, wValue, wIndex, wLen, desc);
}
```

GET_STRING_DESC: The string names associated with the device are requested by the Host in the following way:

```
//-------------------------------------------------
    // Get String Descriptors
    //-------------------------------------------------
    pStr = (pStrDesc)DBUF;
    if (!GetDesc(uAddr,STRING,0,4,DBUF))                    // Get string language
        return FALSE;
    strLang = pStr->wLang;                                  // get iManufacturer
String length
    if (!GetDesc(uAddr,(WORD)(uDev[usbaddr].iMfg<<8)|STRING,strLang,4,DBUF))
        return FALSE;                                       // get iManufacturer
String descriptors
    if (!GetDesc(uAddr,(WORD)(uDev[usbaddr].iMfg<<8)|STRING,strLang,pStr-
>bLength,DBUF))
        return FALSE;
```

GET_CONFIG_DESC: The configuration, interface, device class, protocol, and endpoints associated with the slave device are requested by the host using similar methods.

SET_CONFIG_DESC: This request indicates the completion of standard enumeration request by a Host to a Slave device.

All the Standard EP0 enumeration requests described above are processed using the following important functions.

VendorCmd(): This functions fills the SET UP packet fields with the provided info and passes it on to epoxfer() function.

ep0Xfer(): This function calls *usbXfer()* to setup and initiate any host requests as necessary by passing information like USB address, payload size, SETUP request types and buffer location of returned data (should there be any). Basically, a USB request will consists of three stages, namely the setup stage (SETUP token with request DATA0), data stage (IN/OUT token with DATAx) and status stage (IN/OUT token with null DATA0). Actual USB traffic on the D+/D- lines is initiated by the routine *usbXfer()*.

***usbXfer()*:** This is the core of all USB data and control transactions process. It writes to appropriate registers of SL811HST to initiate a USB transaction as required, be it a write (SETUP/OUT + data) or a read (IN). It also handles low-speed transaction through a hub by appending a PREamble token for any request that goes down all the way to a low-speed device attached to a hub. After each host request is sent, it will wait for an acknowledgement from the slave device by means of an interrupt. It will then determine the type of response from the slave device and terminate as necessary. If there is a request for multiple data from device that is greater than the maximum endpoint zero payload size, it will need to re-arm the SL811HST to grab the next set of data from device until all have been received. For sending data to the device, simply store data into the buffer, give SL811HST the start address of this buffer, set the data length and arm the SL811HST to start USB's SETUP/OUT data transaction.

USB Data transfers: *DataRW()* function is similar to *ep0Xfer()*, except that it is used for data transfer on Bulk, Interrupt and Isochronous endpoints. By specifying the USB device address, endpoint address, maximum payload size, data length and buffer address, we are able to initiate any data transaction, including IN to or OUT of, the slave device. Of course, you will need to write to the buffer if you are doing an OUT data transfer.

```
int DataRW(BYTE usbaddr, BYTE epaddr, WORD wPayload, WORD wLen, BYTE *pData)
{
    xdata BYTE pid = PID_OUT;

    if(epaddr & 0x80)       // get direction of transfer
        pid = PID_IN;

    if(usbXfer(usbaddr,epaddr&0x0F,pid,0,wPayload,wLen,pData))
        return TRUE;

    return FALSE;
}
```

4 USB Host Stack Support

Cypress supports a number of host stack implementations for the SL811HST by providing a host controller driver. Supported operating system stacks include VxWorks, WinCE, and Linux. Cypress also offers a SL811HST development kit with host firmware examples. Some implementations may be available on the Cypress web site. Others not listed on the web site are available from Cypress USB applications support upon request.

5 Summary

The SL811HST can be used as a versatile and full-featured embedded USB host controller. The combination of a standard signaling interface and simple control registers allows the SL811HST to be integrated with a small or large scale embedded USB host stack. For further questions and assistance please contact Cypress USB applications support.

Document History

Document Title: AN1215 - Basic Embedded Host Using the SL811HST

Document Number: 001-16953

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1345183	YIS	08/03/2007	New Spec. The application notes template has undergone a complete make-over so that information is presented in a clear and better organized manner. The number of styles has been reduced and the styles are defined on the template so that a new user can understand what the style is going to look like when applied. Author and other pertinent information is on the first page, but is used only if appropriate and is not required for all application notes.
*A	3158982	NMMA	01/31/2011	Updated title. Added application note abstract, updated template. Added CY3662 development kit board snapshot. Explained CY3662 development kit host mode firmware example.
*B	3719080	NMMA	08/21/2012	Updated template.
*C	4870161	LIP	09/10/2015	Updated template Sunset review
*D	5810366	AESATMP9	07/11/2017	Updated logo and copyright.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Internet of Things	cypress.com/iot
Memory	cypress.com/memory
Microcontrollers	cypress.com/mcu
PSoC	cypress.com/psoc
Power Management ICs	cypress.com/pmic
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless Connectivity	cypress.com/wireless

PSoC® Solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP| PSoC 6

Cypress Developer Community

Forums | WICED IOT Forums | Projects | Videos | Blogs | Training | Components

Technical Support

cypress.com/go/support

All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

©Cypress Semiconductor Corporation, 2007-2017. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, WICED, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.