

PSAS Isochronous USB on LPC2148

PSU CS506-Massey

Dave Camarillo and K Wilson

Table of Contents

Summary.....	1
Development Environment.....	1
How the Project Progressed.....	1
Discoveries.....	2
Results.....	2
Other Documentation.....	2
Appendix A: Race condition in LPC2148 RD_EN.....	3

Summary

The NXP (*née* Phillips) LPC2148 is an ARM7TDMI-S core combined with various system peripherals on a single die. One of these peripherals is a USB interface and protocol device.

Using USB in isochronous mode is attractive to the Portland State Aerospace Society's (PSAS) design for the avionics system currently under development. This goal of this project was the development of the firmware for the LPC2148 so this system could be used as sensor and operational nodes and communicate data and control throughout the avionics system using the USB bus in isochronous mode.

Development Environment

As PSAS has an open source/open hardware/FLOSS philosophy, this project used open source tools and software. We used the Gnu Compiler Collection to cross compile from a linux x86 environment to arm executables and programmed the flash on the lpc2148 using a JTAG device. We also used the open source IDE 'Eclipse' with embedded debugging plugins.

We started by downloading the LPCUSB open source software which did not have isochronous mode USB support and began to build on what already existed.

Neither of us knew much about the USB specification or any typical implementation standards, though we have both worked with other bus protocols in the past. As we soon learned the USB specification is baroque and often poorly implemented. (Reference my Belkin(R) USB hub, which can't register attached isochronous endpoints.)

How the Project Progressed

During the project, we came to understand many of the details and nuances of the USB protocol, including performance characteristics, operational modes and nuances, and how to utilize the bus.

Of particular interest we had to do an end-to-end implementation to get our design to work (linux host to device firmware). To make this happen, we utilized USBFS, made available by the linux kernel, came to an understanding of how the kernel was servicing our USB requests and how they were getting sent over the wire and scheduled.

We then had to come to an understanding of how the USB protocol engine on the LPC2148 worked and how it needed to be programmed in order to communicate with the linux host. Once we had basic end-to-end communications working we were in a position to enhance the existing LPCUSB code base by implementing isochronous mode transfers, and the DMA flavor of those isochronous mode transfers.

Discoveries

During the development process, we ran into numerous problems, all revolving around the nuances of the USB protocol engine on the LPC2148, as well as how the host OS interacted with the device. There were cases where the insertion of a "nop" instruction would "magically" fix a problem (*see Appendix A: Race condition in LPC2148 RD_EN*), and cases where the semantics and intended usage contexts were very poorly defined. The documentation did contain most of the general information we needed, but it is bereft of many salient details and so poorly edited that it is very difficult for the developer to develop a cohesive understanding of how the USB peripheral works and how it is intended to be used.

Results

The project roughly went as we expected it to, based on the plan we had created. We achieved almost all our primary goals (with the exception of integration into with FreeRTOS).

- Implemented isochronous mode, with DMA transfers, at full theoretical speeds (measured with our linux host test application).
- Created a set of simple, bare-bones sample code examples that a prospective ISOC developer might want to copy/modify for their purposes.
- Created a proof-of-concept node that read from the ADC on our development board and emitted those readings to the console of our development host.

Other Documentation

This project used ikiwiki collaborative documentation, which is still active at the following website:

<http://psas.pdx.edu/lpcubsisochronous/>

Details of the LPCUSB project we referenced are here:

<http://wiki.sikken.nl/index.php?title=LPCUSB>

The user manual for the lpc2148 is found here:

<http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc2141.lpc2142.lpc2144.lpc2146.lpc2148.pdf>

We used the Olimex lpc2148 development board. Here is a link:

<http://www.olimex.com/dev/lpc-p2148.html>

We used the Olimex JTAG On Chip Debugger to program and debug the board:

<http://www.olimex.com/dev/pdf/ARM-USB-OCD.pdf>

Appendix A: Race condition in LPC2148 RD_EN

It seems clear that the LPC2148 has a race condition in reading from the USBRxPLeN register in isochronous mode. Frame interrupts happen every mS. The PKT_RDY and the DV flags of the USBRxPLeN register must be checked to know if there is data in the EP (Endpoint) Buffer.

Either the data from USBRxPLeN does not reach the cpu in time for the test

```
if( (USBRxPLeN & PKT_RDY) == 0 )
```

Or the RD_EN signal is not registered (latched) fast enough. The cycle of set RD_EN, check DV is too fast without the 'nop' delay. Spinning on the DV flag is not a solution, since there may not be data valid during this interrupt, or any future interrupts and we would never exit the handler.

```
// set read enable bit for specific endpoint
USBCtrl = RD_EN | ((bEP & 0xF) << 2);
```

Here is the code snippet:

```
// set read enable bit for specific endpoint
USBCtrl = RD_EN | ((bEP & 0xF) << 2);

// here is where we insert 'nops' to test race condition on
// set RD_EN

// remove this what happens? - no data valid seen-ever.
asm volatile("nop\n");

// Time delay from USBRxPLeN to cpu...

// If USBRxPLeN is used directly: failure always, without nop
// PATH: USB->AHB TO APB BRIDGE -> AHB BRIDGE -> ARM7

if( (USBRxPLeN & PKT_RDY) == 0 ) {
    USBCtrl = 0; // make sure RD_EN is clear
```

```

//      DBG("dwLen is: %u. PKT_RDY is %u\n", dwLen, PKT_RDY);
return(-1);
}

// packet valid?
if ((USB RxPLen & DV) == 0) {
//      DBG("dwLen is: %u. DV is %u\n", dwLen, DV);
USB Ctrl = 0; // make sure RD_EN is clear
return -1;
}

```

LPC214x_01_UM.book

File Edit View Go Help

Previous Next 212 of 354 150%

Philips Semiconductors

UM10139

Chapter 14: USB device controller

Table 201. USB Receive Packet Length register (USB RxPLen - address 0xE009 0020) bit description

Bit	Symbol	Value	Description	Reset value
9:0	PKT_LENGTH	-	The remaining amount of data in bytes still to be read from the EP_RAM.	0
10	DV	-	Non-isochronous end point will not raise an interrupt when an erroneous data packet is received. But invalid data packet can be produced with bus reset. For isochronous endpoint, data transfer will happen even if an erroneous packet is received. In this case DV bit will not be set for the packet.	0
		0	Data is invalid.	
		1	Data is valid.	
11	PKT_RDY	-	Packet length field in the register is valid and packet is ready for reading.	0
31:12	-	-	Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined.	NA

8.5 USB Transmit Data register (USB TxData - 0xE009 001C)

For an IN transaction, the CPU writes the data into this register. This data will be transferred into the EP_RAM before the next writing occurs. There is no interrupt when the register is empty. The USB TxData is a write only register.

Table 202. USB Transmit Data register (USB TxData - address 0xE009 001C) bit description

Bit	Symbol	Description	Reset value
31:0	TransmitData	Transmit Data.	0x0000 0000

8.6 USB Transmit Packet Length register (USB TxPLen - 0xE009 0024)

The software should first write the packet length (\leq Maximum Packet Size) in the Transmit Packet Length register followed by the data write(s) to the Transmit Data register. This register counts the number of bytes transferred from the CPU to the EP_RAM. The software can read this register to determine the number of bytes it has transferred to the EP_RAM. After each write to the Transmit Data register the hardware will decrement the

