PLX USB338x Firmware Porting Guide

Version 1.0

September 2011

Revision History

Revision	Date	Comment
1.0	September 2011	Initial Release

1 Contents

1. Introduction

1.1 About this Guide

This document provides essential guidelines for porting PLX RDK firmware samples to the required target platform. While this guide uses the simplest USB application class from the RDK software package as an example USB device, the same process can be applied to any firmware application. Each USB application class contains sources designed for portability, a hardware abstraction layer that is simple to port and built-in debug features that will allow PLX support team to help you along the way.

1.2 Getting Familiar with the USB338x RDK

The PLX RDK with evaluation board is an ideal environment for familiarizing the USB 338x. The "Getting Started" document provides an excellent introduction of the USB basics and PLX software. Apart from the basic guide, it is recommended to try building and testing the PLX Transfer and Loopback USB class projects that comes bundled with the RDK Platform development. This will validate both the integrity of the delivered PLX source code and hardware.

For best results, it is recommended to develop a working model of the final device using the PLX RDK evaluation board. Unlike a newly designed and untested prototype board, the PLX RDK evaluation board is validated and stable hardware. This allows software engineers to concentrate on firmware development without worrying the underlying hardware. It also saves precious engineering man-hours because development can start without any delay of prototype board fabrication. Even after the prototype hardware is ready, the RDK evaluation model can still serve as a functional basis of comparison.

Assumptions: This guide assumes that the RDK sources are copied to a development machine where "<RDK_INSTALL_PATH>" refers to the installed RDK folder.

1.3 PLX Board Design Review Service

PLX strongly recommends all customers to submit their board schematics and layout to www.plxtech.com/support for schematic review before any prototypes are built. For the benefits of PLX customer, PLX will only need the USB portion of the schematic. PLX provides this service free-of-charge to RDK customers and it can save you time and money by catching board design problems early in the development cycle.

2 Building PLX Firmware using Target Platform compiler

In this section, the Transfer class project will be used as a porting example. After the Transfer class project is ported to the targeted CPU, the targeted platform will be able to perform simple USB transfer through USB 3.0 endpoints. In order for that to happen, the existing source code must be compiled in the target CPU's compiler before the firmware can run on the target platform. Below are the general steps taken to compile the firmware. For more details, please refer to the respective, platform documentation.

2.1 Creating a Project File

Create a new project file or makefile for the Transfer project using the target platform build environment. Add all of the ".c" files from the following directories into the project:

< RDK_INSTALL_PATH >\Sources\Src\Usb\Device\NcApps\Transfer\

< RDK_INSTALL_PATH >\ Sources\Src\Usb\Device\NcChips\Net2282\

2.2 Compile Time Switches

PLX source files had several compile time switches to control inclusion and exclusion of code sections. These switches should be used appropriately for individual platform settings. The following compile time switches can be found in NcOptions.h.

_NCDEBUG: Enables print messages and other debugging functions, such as the ASSERT macro. Assert statements check that specific runtime expectations are met. If they are not met, then the ASSERT macro can be set up to print a message. In some development environments, it is possible break the execution of the code and handover control to a debugger which allows the developer to perform post-error analysis.

_NCDEBUG should be set to TRUE only during early development. For optimized production builds, _NCDEBUG should be set to FALSE.

_NCHISTO: Enables the HISTO() trace logging macro. HISTO is a highly efficient way to log and trace the execution of the PLX source code. Each call to HISTO adds a sixteen byte entry to a circular History buffer which marks a point in the code. The first four bytes of each entry make up a unique string index that can be used to look up the entry's location in the source code. The twelve remaining bytes of each entry logs three important values that are critical to that execution point. HISTO truly shines when you require support from PLX. Our support staff makes use of History logs as a source of information when resolving support inquiries.

HISTO should be enabled in development builds and for debugging problems in timing critical situations. Production builds should set _NCHISTO to FALSE.

_NCSTATISTICS: Enables code that tracks endpoint transfer performance and efficiency. This should be enabled for Debug build only.

2.3 Include Directories

Add the following directories to the project file as include directories:

< RDK_INSTALL_PATH > \Sources\Src\Usb\Device\NcApps\Transfer\

< RDK_INSTALL_PATH >\ Sources\Src\Usb\Device\NcChips\Net2282\

< RDK_INSTALL_PATH >\ Sources\Inc\

2.4 Compile the Project

The project is now ready for compilation. Even though, PLX had made source code as portable as possible, different platform compiler can have various proprietary requirements. Minor adjustment is expected to be made during the porting process. With a properly configured project file, the RDK source code should be compiled with very little or no error. However, if you think a major portability problem exists that could affect a number of C compilers, feel free to report it at www.plxtech.com/support.

3 Customizing Sources for Your Hardware

3.1 Driver Entry Point

Like all C program, the main entry point of a program begins with the **main** function in Main.c file. When the target platform is powered up, the following critical tasks must be performed:

1. Make a function call to **System_InitializeSystem** function where the PLX base address, interrupts and other system-specific aspects of the device are initialized.
2. Call the application **OneTimeInit** function to initializes the USB application data structure, related API, and USB338x. Do note that, for the Transfer project, it has already been done.

Also included in **main** function is an infinite loop that polls the keyboard for input. If the target platform has a console or UART interface, this input polling maybe useful for program interaction. Otherwise, it is not essential to the operation of the Transfer project. However, for projects that rely on the keyboard for input, such as the Keyboard and Mouse USB classes the infinite loop structure is necessary for the operation to be completed correctly.

3.2 System Initialization

There are two copies of **System_InitializeSystem** function in **System.c**. The second copy is an empty stub and is enabled when _NC_RDK_AND_WINDOWS is FALSE. Fill in the **System_InitializeSystem** function to process the following:

1. Reset the USB338x controller. Since the USB338x is a PCIe chip, it is normally not necessary and sometimes harmful to reset the device. Most PCIe systems rely on the system BIOS to initialize a PCIe adaptor configuration. In most cases, it is not necessary to reset the USB338x from the firmware project.

2. Setting up the USB338x device Base Address. In most cases, the memory mapped base address of the USB338x is static and is derived from the PCIe platform specific manner. Initialize the global variable, NETCHIP_BASEADDRESS in System.c, so that the entire source tree can read the USB338x Base Address 0.
3. Initialize the system level interrupt. There are two parts to this task. First of all, any CPU specific interrupt enable connected to the USB338x INTA# pin must be set. Second, the interrupt handler function that traps USB338x interrupt must call **Nc_InterruptHandler** function in NcFwApi.c.
4. Lastly, returns the status of the operation.

4 Interrupt Handling

The **System_InterruptController** function provides the basic controls such as enable, disable and returning statuses of an interrupt controller connected to PLX INTA# pin. Usually, all function stubs need to be completed for target platforms. However, carefully crafted code may not need this interrupt controller interface: If it can be guaranteed that all API calls are made from an interrupt context AND no API calls can be interrupted with other API calls, it should be safe to remove all references to this interrupt controller function.

4.1 Memory Allocation and Thread Handling

There are several function stubs such as NcMalloc(), NcFree() and NcSleep() available for platform customization. These stubs facilitate easy optimization of the project. For optimized code, platforms tend to use non-standard function name for these capabilities. For each of these function, the standard corresponding C function will be malloc(), free and sleep().

4.2 USB338x Hardware Abstraction Layer

Most PCIe platforms use memory mapped I/O to read and write to external PCIe adaptors. In those instances, NcHal which is the Hardware Abstraction Layer does not require any customization. However, if the target platform does not use memory mapped I/O, then these changes must be made:

1. Set NET2282_DIRECT_ACCESS in NcOptions.h to FALSE. This switches the NETCHIP_READ and NETCHIP_WRITE macros in NcHal.h to the functions NcRead32() and NcWrite32().
2. Customize NcRead32() and NcWrite32(). Modify NcRead32() and NcWrite32() in NcHal.c to suit your CPU I/O scheme. NcRead32() takes a USB338x register offset, and returns the byte value read from that offset. NcWrite32() takes a register offset and a value and writes that value to the register.
3. Adjust accesses to EP_DATA. For performance, reads from and writes to EP_DATA do not utilize the NETCHIP_READ and NETCHIP_WRITE macros. So you will have to globally search for all the EP_DATA reads and writes and adjust them to fit your CPU architecture.

4.3 Memory Mapped I/O

At this point of time, the major system level source code and customization are completed. The next few sections provide guidelines on how to debug the USB device in the target platform to get Transfer class fully operational.

4.4 Verify USB338x Read/Write Access

Using the RDK API, the **NcDev_AreYouThere** function will perform a read and write test on the USB338x to prove that chip access is reliable. If **NcDev_AreYouThere** function returns TRUE, then the verification is complete. If the function fails, below are some steps to debug the issue.

1. Does **System_InitializeSystem** function return successfully? If something went wrong in the system level initialization, then chip access may not work. In particular, the global base address pointer to the USB338x, NETCHIP_BASE_ADDRESS, must be initialized correctly.

2. Is the ApplicationObject->**OneTimeInit** function called successfully from main function? If ApplicationObject is not initialized correctly in main function, then the call to ApplicationObject->**OneTimeInit** function has no chance of working. Verify that it is called, and follow the execution to **NcDev_AreYouThere** function.
3. Getting insight to the USB can be helpful. Therefore, a logic analyzer or PCIe bus analyzer might provide a clue of what is going wrong. These tools are very useful for debugging both hardware and software.

5 Verify Interrupt Calling and Handling

If register access is reliable, then USB338x initialization is almost guaranteed to complete successfully. The USB338x is ready to generate an interrupt on any USB activity. Connect the USB Host and Device and verify that **Nc_InterruptHandler** function gets called. If it doesn't, then check these points in the system:

1. On USB cable connection, does the USB338x INTA# pin assert? If the INTA# pin is not asserting, this indicates that USB338x initialization was not successful. A simple interrupt to detect is the VBUSInterrupt. If VBUS_INTERRUPT_ENABLE in the PCIIRQENB1 register is set and the USB cable is plugged in, then the INTA# pin should assert and the VBUS_INTERRUPT bit should be asserted in the IRQSTAT1 register.
2. Does your Interrupt Handler function get called? If your interrupt handler never gets called, then this could indicate that the interrupt enables in your CPU have not been set correctly.
3. Does your Interrupt Handler call Nc_InterruptHandler()?

6 USB Enumeration

With interrupt handling in place, the Transfer project is ready to perform USB Enumeration. USB Enumeration is the process which the USB Host goes through in identifying the recently connected device, probing its configuration, and loading a driver for its operation. It involves several types of Setup requests that are performed on Endpoint 0, the Control Endpoint. Here are several hints on getting your device to enumerate:

1. HISTO function is an extremely useful tool in debugging enumeration. It provides the call tracing functionality which keep tracks of the Enumeration process.
2. Does a Setup Packet Interrupt ever occur? If a Setup Packet Interrupt never occurs, then this could indicate a hardware failure. Your board layout could be causing USB signal quality problems such that the USB338x cannot detect a Setup Packet. Just to be sure, check that SETUP_PACKET_INTERRUPT_ENABLE is set in IRQENB0. If it is correctly set and there is no interrupt, then there could be a layout or schematic issue.
3. If you have access to a USB Bus Analyzer, use it.

7 Bulk Transfer

Now that the USB device in the target platform enumerates, it is time to move from control to bulk endpoint testing. Transfer class is a simple data source-sink application created by PLX. The USB host will sent data to using Bulk Out endpoints and supplies meaningless data on its Bulk In endpoints. The USB host application is supplied by PLX known as WinMon application that can be found also on the RDK. The WinMon application is used to test basic transfer functionality. Use “OUT” and “IN” to initiate transfers. Type “? out” and “? in” (without the quotation marks) in WinMon for help.

Note: Transfer initiates transfers of 0x10000 bytes for any Bulk In endpoint. And, it terminates every In transfer with a short packet. Thus, expect a Zero Length Packet (ZLP) for every 0x10000 bytes of IN data.

8 Loopback Testing

With basic bulk transfers working, the device can be further tested with Loopback transfers using the Bulk endpoints. Loopback transfers route data from Bulk Out to Bulk In endpoints hence testing both channel of the software and hardware. Porting the Loopback project is trivial because the core firmware is the same for Transfer class. The loopback test is available either using LoopHost or WinMon running on the USB host. In WinMon, issue the command “loop” to initiate testing. Type “? loop” (without the quotation marks) in WinMon for help regarding its parameters. For LoopHost, simply launch the GUI application and click on the Start button to run the loopback test. At this point of time, it might be a good project milestone to perform an over-night test to check out the reliability of both the hardware and software before proceeding further.

9 Device Integration

At this point, the USB section of your hardware and firmware should have been thoroughly tested. You can now integrate the other parts of your device that actually produce or consume the data that goes over the USB. Here are some hints to keep in mind during this process:

1. Kicks off the project development with a PLX RDK Evaluation board and firmware API software. Since MS Visual Studio and Windows are readily available, it will provide a low-cost starting point with feature rich development workspace. Many embedded development environments will take days or even weeks to setup before any code can be compiled. PLX highly recommend building up a new project on the RDK platform that enumerates under the exact descriptors and configuration your device would use. Take this model as far as it can go. In certain situations, it is possible to simulate hardware on the RDK Platform. For instance, the RDK MassStorage project simply uses an array of bytes to act as a virtual disk memory. Once the device model is working on the RDK, it can be ported it to any target platform.
2. Don't forget about HISTO function. Feel free to sprinkle HISTO function entries throughout the application level source code. Think of HISTO function as a logic analyzer for the software. Entries can be logged quickly at runtime, but printed and analyzed later.

10 Conclusion

PLX Engineering Team has worked hard to make this document complete and useful. However, the team recognizes that you may still run into problems and difficulties. If you cannot resolve issues related to the distribution or this document, please contact PLX:

- On the Web: http://www.plxtech.com/support
- Subject: USB338x Firmware Porting Guide
- Please include the following information:
- Distribution revision level
- Concise description so that we can recreate the problem
- 'Cut and paste' any supportive text output (add helpful explanations)

Legal Disclaimer

Information in this document is provided in connection with PLX technology, Inc. products. No license, express or implied, by estoppels or otherwise, to any intellectual property rights is granted by this document. Except as provided in PLX technology's terms and conditions of sale for such products, PLX technology assumes no liability whatsoever, and PLX Technology disclaims any express or implied warranty, relating to sale and/or use of PLX Technology products, including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right.

This document provides technical information for the user. PLX Technology, Inc. reserves the right to modify the information in this document as necessary. The customer should make sure that they have the most recent datasheet version. PLX Technology, Inc. holds no responsibility for any errors that may appear in this document. Customers should take appropriate action to ensure that their use of the products does not infringe upon any patents. PLX Technology, Inc. respects valid patent right of third parties and does not infringe upon or assist others to infringe upon such rights

All information contained herein is subject to change without notice.