Xilinx Answer 51950

Tandem PCIe Second Stage Bitstream Loading across the PCI Express Link

Important Note: This downloadable PDF of an Answer Record is provided to enhance its usability and readability. It is important to note that Answer Records are Web-based content that are frequently updated as new information becomes available. You are reminded to visit the Xilinx Technical Support Website and review (Xilinx Answer 51950) for the latest version of this Answer.

Introduction

Tandem PCI Express implements a two stage configuration methodology. The first stage configuration memory cells that are critical to PCI Express operation are loaded via a local PROM. When these cells have been loaded, an FPGA Startup command is sent at the end of the first bitstream to the FPGA configuration controller. The partially configured FPGA then becomes active with the first-stage bitstream contents. The 1st stage containing a fully functional PCI Express port responds to traffic received during PCI Express enumeration while the second stage is loaded into the FPGA. The second stage consists of the user specific application and is loaded via the PCI Express link into the Internal Configuration Access Port (ICAP).

The first part of this document describes the general Tandem Software Flow.

The second part of this document describes the software device drivers for the Xilinx Fast Partial Configuration (FPC) solution for PCI Express (PCIe) designs implemented using 7 Series devices.

The Xilinx FPC mechanism permits a PCIe card employing a 7 Series device as its PCIe endpoint controller to enumerate to the PCIe root complex using a partial bitstream which is readily loaded within the minimum time between power-on and readiness defined by the PCIe specification. The stage2 programming file may then be loaded over the PCIe bus, without restriction on the amount of time required to do so. Two example device drivers are provided with the Xilinx FPC solution, one for Windows and another for Linux, which fulfill this role. These drivers are loaded and bound to devices using the FPC scheme, and then expose a software interface for enumeration and stage2 loading. An example application is also supplied for each driver, demonstrating the stage2 loading of an FPC device within each environment.

For better illustration of Tandem flow for the readers, the flow has been described at the end of this document in reference to KC705.

Tandem Software Flow

Tandem RTL Design

The Tandem PCIe design implements an additional module which is referred to as the FPC module. The FPC module is responsible for reception of data off the PCI Express link and transmitting the data into the Internal Configuration Access Port (ICAP). By design, the FPC will take data from Memory Write (MWR) requests and send it to the ICAP. The Tandem FPC implements an asynchronous FIFO to handle buffering of received data. The FIFO depth is configured with the assumption that data will be received via Memory Writes with 1DW of payload. The FPC will complete all Memory Read (MRD) requests with a 1 DW Completion w/ Data (CPLD), and the CPLD payload will always be 32'h0. All other packets will be purged from the core and no additional processing will be performed.

The FPC module only responds to Memory accesses targeting BAR 0 of the Integrated Block for PCI Express. The FPC module considers all Memory Write request targeting BAR 0 to be configuration data destined for the ICAP.

Tandem Configuration

Loading the first stage

Depending on the programming methodology, load the first stage bitstream into the FPGA or Flash. Boot the system and verify that the device was recognized. Also ensure that the device has been allocated a memory region at BAR 0.

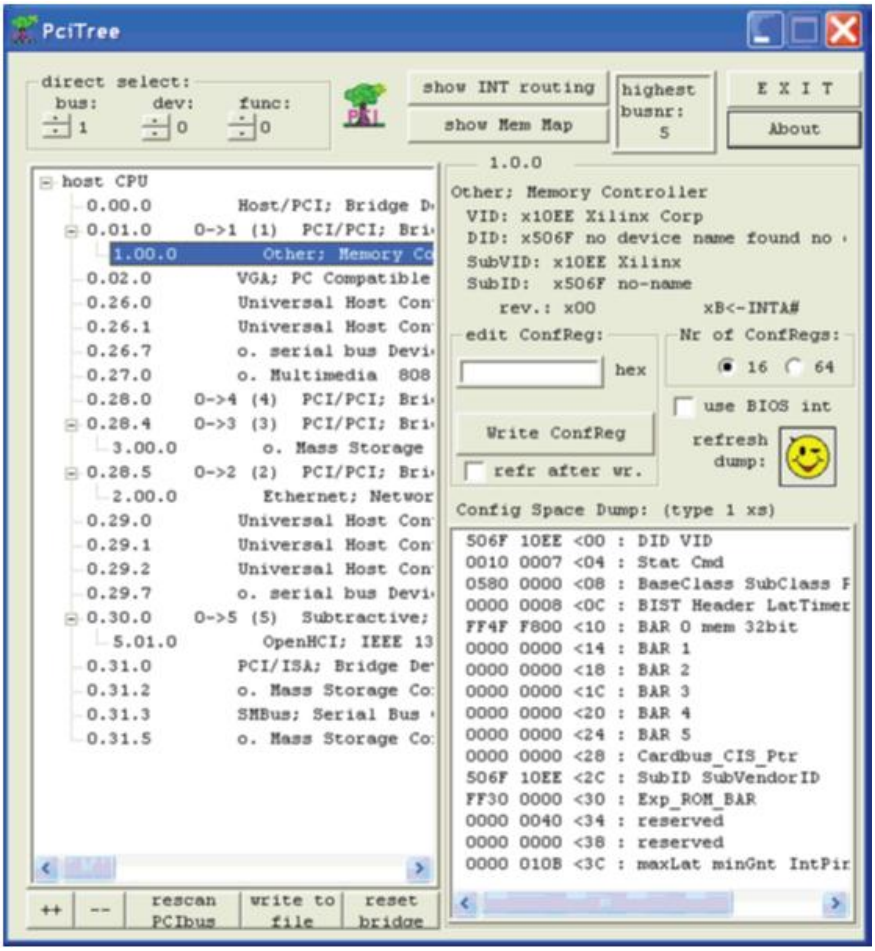
Linux Based OS:

Linux has built in support for viewing PCI devices and their configuration space within a system.

>sudo lspci –vvv

Windows Based OS:

A number of utilities are available to verify the first stage loaded successfully. One common utility is PCITree.



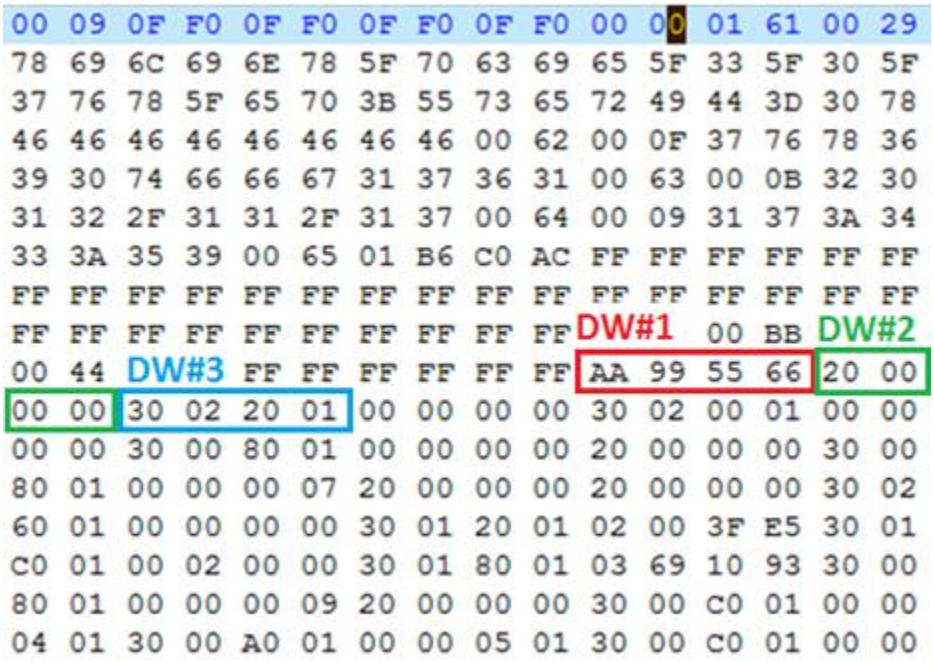
Note: Depending on the core configuration, the value seen in the screen-grab above may differ.

Configuring the second stage

Once the first stage is confirmed to be operating as normal, software is responsible for correctly formatting and transmitting the second stage bitstream to the FPGA. All data must be transmitted to the FPGA via Memory Write Requests with a 1DW payload. The address of the write request should always target offset 'h0 of the endpoints BAR 0 memory region.

The ICAP found in the Tandem PCIe design scans for the sync word before starting configuration of the second stage. The sync word (32'hAA995566) should be the first DW sent to the FPGA. Once sent, the rest of the bistream data must be sent sequentially to complete configuration of the second stage.

Below is an example snippet of a bitstream viewed in a hex editor. The first three dwords sent to the FPGA are highlighted. The .bin programming file is algned to this sync word and should be used ot load the stage2 portion of the design. The .bit programming file is NOT aligned to this sync word and should NOT be loaded over the PCI Express interface.



Once the second stage (stage2) has loaded, the DONE indicator will assert and remain asserted.

One common method for transferring data to the device is via a PIO capable kernel mode driver and associated user mode application.

Tandem Drivers

Linux Environment

The Linux FPC driver is specifically targeted at version 3.2.0 of the Linux kernel, although the code is easily back-ported to prior kernel versions if necessary. An I/O control (IOCTL) mechanism is employed for the communication between user space and the driver code.

Miscellaneous Driver

The device driver is implemented as a "misc" character device driver, which uses the major number allocated by the Linux kernel for "miscellaneous" devices. This eliminates the need to define a unique major number for the driver; this is important, as a conflict between major numbers has become increasingly likely, and use of the misc device class is an effective tactic. Each probed device is dynamically assigned a minor number, and is listed with a directory entry within the sysfs pseudo-filesystem under /sys/class/misc/.

PCI Enumeration

The driver registers itself with one or more combinations of Vendor and Product ID. During PCI device enumeration, these and other values are read from each device's configuration ROM for use in plug-and-play driver detection. Listing 1- depicts the PCI device table for the example driver, registering it for the vendor ID assigned to Xilinx (0x10EE), and the product ID used within the example FPGA design on the KC705 board (0x7024). The device table may be used to additionally filter based upon device class; in this example, use of the PCI_ANY_ID value indicates that this driver does not require any particular value for these fields.

```
/* Device table to register the driver for */
static DEFINE_PCI_DEVICE_TABLE(fpc_pci_tbl) = {
  {    XILINX_VENDOR_ID,    PCI_FPC_PRODUCT_ID,    PCI_ANY_ID,    PCI_ANY_ID,    0,    0,
CARD_XILINX_KC705},
  { 0, }
};
MODULE_DEVICE_TABLE(pci, fpc_pci_tbl);
```

Listing 1 – PCI Device Table for Linux Driver

During kernel boot, the driver is invoked for any located PCI devices which match against the table entries, which are passed in the call to `pci_register_driver()` made in the driver module's "init" routine.

This implementation is intended to serve as an example; therefore, it is somewhat monolithic to simplify the code. It is not intended that a single device driver be continually extended to support customer designs; in the simplest case, the FPC code may be copied into the structure of a standalone driver, with its own appropriate PCI device tables. However, to incorporate the FPC functionality into multiple, disparate drivers with unique additional functionality, the FPC code may be factored out into a separate kernel object and dispatched to by client drivers.

I/O Control Operations

The device driver makes use of the `ioctl()` system call to allow userspace to control the device. The definitions for the `ioctl` command codes are defined in `<linux/xilinx_pci_fpc.h>`, and are described in Table 1:

Command Code	Parameter Type	Description
`IOC_GET_BOARD_ID`	`struct fpc_board_id*`	Returns the PCI vendor and device codes in the structure pointer
`IOC_INIT_RECONFIG`	`uint32_t`	Initiates a device reconfiguration, with the specified number of configuration data blocks
`IOC_CONFIG_BLOCK`	`struct fpc_data_block*`	Loads the next block of configuration bitstream data to the device

Table 1 - I/O Control Codes

If the functionality of the example driver is factored out into a separate kernel source module for use by multiple drivers, the `ioctl()` function of each "client" driver may simply invoke the shared FPC `ioctl()` code with the command and void* parameter if it does not recognize the command code as one of its own, device-specific commands.

Two structures are defined for use within `xilinx_pci_fpc.h`:

```
struct fpc_board_id {
  uint16_t vendor;
  uint16_t device;
};
```

Type `struct fpc_board_id` is a simple structure used to aggregate the PCI vendor and device ID of a device which was probed. This allows multiple device drivers to use the same misc device prefix and driver code, and permit userspace applications to filter out only those devices in which they are interested by ID.

```
struct fpc_data_block {
  uint32_t num_words;
  uint32_t block_words[MAX_CONFIG_BLOCK_SIZE];
};
```

Type `struct fpc_data_block` is used to conduct blocks of configuration bitstream data for the device down to the driver from userspace. The MAX_CONFIG_BLOCK_SIZE defines the size of each block, and the call to IOC_INIT_RECONFIG must be made with the following value for its parameter:

```
cfg_words = ((cfg_bytes + sizeof(uint32_t) – 1) / sizeof(uint32_t))

num_ blocks = ((cfg_words + MAX_CONFIG_BLOCK_SIZE – 1) /
               MAX_CONFIG_BLOCK_SIZE)
```

In the above expression, `cfg_bytes` is the size, in bytes, of the configuration .BIN file.

Sample Application

The sample application is written in C++, uses POSIX-compliant APIs, and is compiled with the GNU Compiler Collection (GCC) g++ compiler.

The application applies an object-oriented framework to the detection and identification of FPC devices in the system, and uses a polymorphic design pattern to create objects that wrap each device. In this manner, devices with completely different user design functionality are able to share the common FPC facility by deriving from a common base class, `XilinxFpcDevice`.

The PosixTestApp project consists of the following files:

File	Description
`XilinxFpcDevice.h` `XilinxFpcDevice.cpp`	Source files for the base class inherited by all FPC device wrappers. Provides facilities for enumerating all FPC devices in the system, filtering by type if desired. Defines an abstract interface for partially-reconfiguring devices, allowing sub-classes to implement their own means for locating an appropriate .BIN file for use.
`KC705_Board.h` `KC705_Board.cpp`	Example subclass specific for the KC705-based FPC demo. The subclass registers for the KC705 demo vendor and product ID used in the example misc driver, and specifies the BIN file for the demo user partition.
`test_fpc.cpp`	Main test application file, locates any KC705 boards in the system by wrapper class type, causing them to be polymorphically reconfigured and made ready for use.

Table 2 – POSIX Sample Application Source Files

The `XilinxFpcDevice` base class encapsulates the details of scanning the `sysfs` filesystem for FPC boards, opening them, and issuing `ioctl()` commands to fetch the PCI IDs for each located device instance. A PluggableFactories design pattern is used; the base class maintains an associative map of board IDs to "creator" objects that are capable of polymorphically creating a derived class instance that "knows" how to wrap boards with each expected vendor and device ID pair. Through this mechanism, subclasses are able to provide the following:

- Details of how and where to locate the .BIN file appropriate for loading the stage 2 programming file to the board, which may be as simple or complicated as the application requires
- A creator instance to register the class to be invoked by the FPC board detection mechanism when boards with its associated vendor / device ID pair are found
- A public interface appropriate for wrapping the design-specific functionality of the user partition design.

As an example, the class abstracting a board used for accelerating image-processing algorithms may have a series of methods which permit images to be loaded, algorithms specified, etc. These are implemented using `ioctl()` commands understood by the board's custom driver, and have nothing to do directly with the FPC mechanism. The ability to locate the board, associate it with the custom wrapper class, and make use of the FPC mechanism itself, however, are all implemented cleanly within the `XilinxFpcDevice` base class, freeing software engineers from managing those details directly.

A Makefile is provided for building the sample application; simply invoking "make" within the source directory produces the `test_fpc` application binary. The application must be run with superuser permissions, as it will create device nodes for detected FPC devices in the `/tmp` directory off of the filesystem root. This may be accomplished using the `sudo` command:

```
sudo ./test_fpc file=<your_design_tandem2.bin>
```

Upon execution, the application will announce the number of boards implementing the FPC driver mechanism,

e.g.: `Located 1 FPC board(s)`

After execution, it will be noted that the KC705 board is running the user partition, blinking one of the user LEDs as an indication of successful configuration.

Windows Environment

The Windows FPC driver is specifically targeted at Windows 7, although it is more generally stated as a driver developed for use with the Kernel Mode Device Framework (KMDF). This object-oriented framework eliminates much of the error-prone manual manipulation of IO Request Packets (IRP) made necessary by previous Windows driver infrastructures (e.g. WDM).

Both the driver and the sample application used to exercise it are built as sub-projects of the same Visual Studio 2012 "solution", named FpcDriver. The device driver is installed using the devcon utility, for which an example batch file is provided. It is also possible to use Visual Studio to provision a target machine and perform automatic deployment; however, this was found to be not always reliable, and is beyond the scope of this document. The sample application is a console application, which may be run from the Windows command prompt.

KMDF Driver

The KMDF driver is built from sources under the FpcDriver subdirectory of the top level solution.

Driver Source

The main body of the KMDF driver source is in the source file FpcDriver.c. Two header files are also supplied: FpcDriverPublic.h, containing constants and types instrumental to the interface between user- and kernel-space, and

FpcDriverPrivate.h, which provides definitions used strictly internally to the driver module. The public header is included by any user application which makes use of the device's interface.

Finally, an INX file is supplied to serve as a template for creation of appropriate driver information files for installation.

Installation and Device Association

Any driver being installed in a Windows environment must be accompanied by an .INF file which provides the metadata necessary for the Windows operating system to associate the driver binary with the appropriate devices, when they are detected; in this case, during PCI bus enumeration. The contents of the INF files for a build of the driver are contained within the file FpcDriver.inx; this file serves as a template from which INF files are generated upon a successful build of the driver package. During generation, details such as the target architecture (e.g. x86 / x64) for which the driver is being built are filled in around the static information in the INX template.

Within the INX file are the lines which direct Windows which PCI vendor / device combinations to associate the driver with when they are detected within the system:

```
; For XP and later
[MSFT.NT$ARCH$]
; DisplayName       Section             DeviceId
; -----------       -------             --------
%FpcDriver.DRVDESC%=FpcDriver_Device,     PCI\VEN_10ee&DEV_7024
```

Once installed along with the compiled driver binary, the device associations are written into the Windows registry, where they are used to match the driver against devices as they are enumerated. In order to specify multiple combinations for which the driver is to be matched, additional lines may be added.

In order to avoid having to sign this sample driver with a digital certificate, it is classified as a “Sample” driver. As a result, when the driver is installed, the devcon utility also installs a device stub. This device will show up in the Windows Device Manager along with any other detected FPC devices, underneath the “Samples” category. However, upon inspecting the devices by right-clicking and selecting Properties, the stub device will report its location as %Unknown%. This device should be removed after driver installation by right-clicking it and selecting “Uninstall”. Be certain to only confirm device removal, and not the driver as well.

Driver Callbacks

The driver conforms to the pattern of a KMDF driver by way of registering callbacks with the Windows operating system. These callbacks provide an event-driven mechanism for the driver to initialize itself, react to the detection of devices which it matches, and respond to I/O request packets (IRPs) coming from user space applications.

Driver Initialization

As with any Windows driver, `FpcDriver` supplies a driver initialization callback, `DriverEntry`, which is executed upon booting of the Windows kernel once the driver has been successfully installed. This callback is marked for insertion into the Windows system registry, for dynamic loading and execution at boot time, by virtue of its prototype being prefixed with the `DRIVER_INITIALIZE` macro.

The `DriverEntry` callback performs one task: creation of a driver object within the Windows kernel, associated with the callback the kernel invokes upon detection of a device which matches the criteria supplied in the INF file during driver installation. A range of driver-specific configuration options may be specified during this step; however, none are required by `FpcDriver`.

Device Addition

The device addition callback, `FpcEvtDeviceAdd`, is associated with the driver during initialization, and is invoked zero or more times during hardware device detection – once for every device instance which is determined to be a match for the driver's PCIe vendor \ device combination(s). In general, this callback is responsible for configuring additional callbacks and policies which are to be used with the newly-detected device, and creation of a data structure, known as the "device extension". The device extension provides all the information necessary for other callbacks to uniquely identify the device and inspect and / or manipulate its perceived state when they are asynchronously invoked by the kernel.

No actual interaction with the hardware occurs at this time; the device addition callback performs the following actions, in the order listed:

- I/O operations are configured to occur with no buffering (direct I/O)
- Plug-and-play / power management callbacks are registered
- An attributes structure is allocated with enough space for the device extension
- A kernel device object is created to abstract the hardware device
- An interface is associated with a globally-unique identifier (GUID) for the driver's devices
- The driver-specific device extension structure is initialized to a known state

The callbacks registered represent the sum total of all the entry points to the driver which are called during the lifetime of the hardware device.

Hardware Preparation

The hardware preparation callback, `FpcEvtDevicePrepareHardware`, is registered during device addition, and is called after the device is added. This callback is responsible for placing the physical hardware into a known initial state, readying it for use. The Windows kernel invokes the hardware preparation callback with a set of data representing all physical resources the hardware is discovered to possess at plug-and-play detection. Resources include memory or I/O mappings over PCIe, interrupts, etc.

The `FpcDriver` implementation only recognizes and makes use of one resource type, a PCI memory range mapped to the single base address register (BAR) within the fast partial reconfiguration hardware. The physical memory space is mapped into the driver's virtual address space as non-cacheable I/O memory for future use. The resulting mapped address and length of the memory region are recorded within fields of the device extension structure, which is located at the beginning of the callback by way of the `Device` object passed in by the Windows kernel, which was created during device addition.

Hardware Release

Since relatively little is performed by the hardware preparation callback, there is similarly not much to be done in the hardware release callback, `FpcEvtDeviceReleaseHardware`. This callback is invoked at system shutdown; while the PCIe backplane does not support hot-unplugging, device drivers on other bus types make use of this callback to perform cleanup operations when a device is physically removed or powered down.

This callback merely unmaps the I/O memory region, using the virtual base address recorded in the device extension by the hardware preparation callback. As before, the device extension structure is navigated to by way of the `Device` object passed from the Windows kernel.

I/O Write

The I/O write callback `FpcEvtIoWrite` is registered during the execution of the device addition callback `FpcEvtDeviceAdd`, specifically during the initialization of the driver-specific device extension.

The callback is associated with an I/O request queue, into which write request IRPs are placed by the driver framework, triggering execution of this callback. The queue's dispatch policy is set to "sequential"; this indicates to the framework that only one request is to be delivered to the driver's queue at any given time, and pending requests are completed before the next is delivered.

Each write IRP made from user space is associated with a variable-length block of partial-reconfiguration bitstream data, which is written to the I/O memory space mapped for the card, causing the partial reconfiguration design to load the data into the user partition of the Xilinx FPGA it hosts. The reconfiguration words are supplied as a Memory Descriptor List (MDL) associated with the IRP. A virtual address to the pointer encapsulated within the MDL is obtained, and the data words are written to the PCIe register base address in a loop. The sequential dispatch policy specified when the queue was created ensures that the IRPs and their associated data are delivered to the driver in the order they were presented by the user application.

Stubbed Callbacks

In addition to the aforementioned callbacks, several additional callbacks are registered at device addition time as placeholders for other callbacks commonly required by drivers:

- `FpcEvtDeviceD0Entry` – Called when the power management system places the device in a "ready to use" state, referred to as D0
- `FpcEvtDeviceD0Exit` – Called when the system takes the device out of the D0 state
- `FpcEvtDeviceFileCreate` – Called when a user application creates a file handle for the device
- `FpcEvtFileClose` – Called when a user application close a device file handle

Sample Application

The sample application provided for testing the device driver is essentially identical to that described for the Linux environment. It structure is the same, and is a console application called `test_fpc.exe` that is invoked from the Windows command console. Its operation is largely the same, scanning for devices which have enumerated and matched the vendor / device pair(s) which are specified for the driver in its INF file.

The majority of the implementation differences are localized to the `XilinxFpcDevice` class; most of the remaining source is identical to that used to test the Linux driver. The APIs used to enumerate, open, and write data to a device within the Windows operating system are significantly different than those used under Linux to accomplish the same task. The object-oriented nature of the test application allows those differences, no matter how significant, to be largely encapsulated within a single class.

Tandem KC705 Example Flow

This section describes how to perform the second stage partial configuration using the example software targeting the KC705 reference board. The flow described below assumes the core was generated with Vendor ID of 16'h10EE and Device ID of 16'h7024. Similar methodology can be used for other reference boards and custom designs but modifications to the software environment may be required.

Step 1: Prepping the second stage Bitstream

For instructions on generating the first and second stage bitstream, refer to the associated core product guide. After creating the first and second stage .bit files, additional .bit manipulation is required on the second stage. The partial configuration flow requires that the start of the second stage bitstream be 32-bit (DW) aligned. Open up the bitfile (.bit) in a hex editor utility. Once opened, the sync word 32'hAA995566 needs to be DW aligned and shifted accordingly. This is can be achieved simply by removing all data up to the sync word. After deleting the bytes, save the file as <somefilename>.bin.

Note: This stage can be skipped when using stage2.bin file that is generated by the Vivado tools.

Step 2: Load the first Stage

Insert the KC705 reference board into a PCI Express slot within the test system. Configure the PROM device with the first bitstream. See the "Programming the Device" section within the product guide for instructions. Manual loading of the first stage bitstream into the FPGA via JTAG can also be performed to load the first stage into the FPGA.

Step3: Booting the System

Power on the system and verify the card is fully recognized. The BIOS will perform device enumeration, allocate the Tandem BAR 0 used to received 2nd stage configuration data, and set up the PCIe configuration space. Verify the presence of the device using "lspci"

>sudo lspci –vvv –d 10EE:

Step4: Compiling and Installing the Software

Compile the Tandem PCIe kernel driver and C++ user space application using the provided makefile.. Execute the following commands to compile the kernel driver and user space application:

Navigate to the "linux_driver" directory and compile the driver using the makefile:
>make all

The result will be a kernel module called "xilinx_pci_fpc_main.ko"

Navigate to the "linux_test_app" directory and compile the user application using the makefile:
>make all

The result will be an executable called "test_fpc"

Load the device driver into the kernel:
>insmod xilinx_pci_fpc_main.ko

Step5: Run the application

>sudo ./test_fpc file=<your_design_tandem2.bin>

The application will indicate that it found the board and load the second stage configuration data. The board LED's will begin operating as normally seen with the standard monolithic implementation.

"Located 1 FPC board(s)"

Revision History

18/12/2012 - Initial release
08/01/2013 – Upated for Windows Environment
09/12/2013 – Added Tandem Software Flow and Tandem KC705 Example Flow