PCIExpress Hot-Plug Mechanism in Linux-based ATCA Control Systems

Adam Piotrowski, and Dariusz Makowski

***Abstract*—PCI Express architecture is a widely used communication bus designed for industrial application. Additionally, according to PICMG 3.4 specification it is a part of ATCA architecture. One of the features offered by PCI Express standard is possibility of replacing the system components without shutting down entire system. In this paper, authors present general overview of Hot-Plug implementation in Linux operating system used in ATCA carrier board.**

***Index Terms*—PCIExpress, Hot-Plug, ATCA, Advanced Telecommunications Computing Architecture, AdvancedTCA, Linux**

I. Introduction

High availability and reliability are the critical parameters for many systems. Non-stop operation and maintainer downtimes of just a few minutes a year are additional coefficients required by many customers. An example of such system is ATCA-based (Advanced Telecommunications Computing Architecture) Low Level Radio Frequency (LLRF) controller, designed for the X-ray Free Electron Laser project (XFEL) - new linear accelerator that will be located in Deutsches Elektronen-Synchrotron (DESY) research center in Hamburg, Germany [1], [2]. To satisfy such high requirements there must be provided a method to repair failures quickly and without machine shutdown. In systems based on PCI Express (Peripheral Component Interconnect Express) bus, Hot-Plug/Hot-Swap [3] solutions are features that support these goals. PCI Express has been designed as a high-speed, serial computer expansion card standard [4]. Hot-Plug/Hot-Swap solution provide methods to replace modules without turning system off, keeping operating system services running correctly after component removal and restarting or shutting down software associated to removed device.

II. Hardware and Software Support for PCI Express Hot-Plug Mechanism

Several hardware and software components presented in Fig. 1 must be implemented to support the Hot-Plug functionality. From software point of view, following elements are required [5]:

- *Hot-Plug Service* responsible for processing commands from operating system and sending it to standardized Hot-Plug Driver,
- *Hot-Plug Driver* that interact with hardware Hot-Plug Controller to accomplished requests,
- *Device Drivers* that support several Hot-Plug specific commands.

The application programming interface for hardware Hot-Plug Controller for the root complex and switch is standardized,

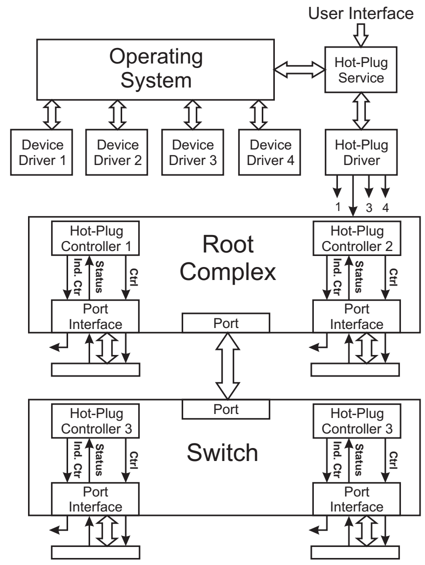
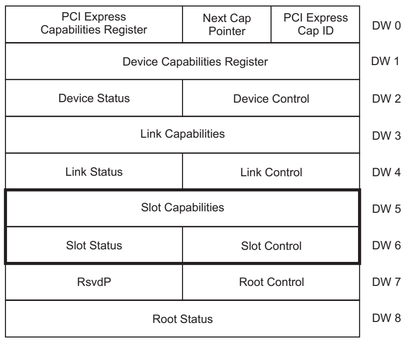


Fig. 1. Hardware and software components required to support Hot-Plug mechanism [5].

therefore any Hot-Plug Driver is able to control behavior of mechanism. Programming interface to PCI Express controller



PCI Express Capabilities Register	Next Cap Pointer	PCI Express Cap ID	DW 0
Device Capabilities Register			DW 1
Device Status	Device Control		DW 2
Link Capabilities			DW 3
Link Status	Link Control		DW 4
Slot Capabilities			DW 5
Slot Status	Slot Control		DW 6
RsvdP	Root Control		DW 7
Root Status			DW 8

Fig. 2. PCI Express Capability Register Set [5].

mainly utilizes Slot Registers located on the offset 5 and 6 in PCI Express Capability register block presented in Fig 2. Slot Capabilities register contains information about functionality offered by the Hot-Plug controller, e.g. support for Hot-Plug operations and surprise device removal without system notification, presence of attention button, attention indicator or power indicator. Slot Control register allows to control the Hot-Plug operations and enables various controller features. For example, it controls behavior of LEDs (Light Emitting Diodes) indicators available in system and allows to enable Hot-Plug interrupts e.g. to indicates Power Fault Detection or

Presence Detect Change. Slot Status register allows to monitor variety of events related to Hot-Plug system. Software can monitor register to determine which event has occurred. Additionally, if interrupt system related to controller is enabled, appropriate interrupt will notify system about Hot-Plug event. Slot Status register contains six bits responsible for indication about detection of following conditions: attention button was pressed, power fault at port was detected by Power Controller, state of MRL sensor was changed, change in the state of slot was detected, last Hot-Plug software command was completed. The two last bits inform about precise state of MRL Sensor (open or close) and Presence Detect (card is installed into slot or not).

Major hardware elements required to support PCI Express Hot-Plug functionality are as follow [5]:

- *Hot-Plug Controller* responsible for processing commands issued by Hot-Plug Device Driver. Each root complex and switch port that supports Hot-Plug functionality has associated one controller, see Fig. 1,
- *Card Slot Power Switching Logic* allows to turn on/off power supply to selected PCI Express slots. Logic is controlled by Hot-Plug Controller and indirectly through Hot-Plug Device Driver,
- *Card Reset Logic* allows to assert or deassert PCI Express Reset (PERST#) signal to specified slot. It is controlled by Hot-Plug Controller and indirectly through Hot-Plug Device Driver,
- *Power Indicator* inform if power is applied to selected slot. Controlled by Hot-Plug Controller and indirectly through Hot-Plug Device Driver,
- *Attention Indicator* inform if Hot-Plug problem or failure occurred. Controlled by Hot-Plug Controller and indirectly through Hot-Plug Device Driver,
- *Attention Button* used by user to notified Hot-Plug software about Hot-Plug request,
- *Card Preset Detection Pins* are located at two opposite ends of PCI Express card slot. Those pins are shorter then rest of the connectors, therefore allows to break-first capability upon card removal.

III. PCI Express Bus Driver

Standard Linux PCI Driver Model allows to install only one driver for one device. The PCI Express may offer multiple services operating independently, like support for Virtual Channels (VC), Advanced Error Reporting (AER), Power Management (PME) or Hot-Plug (HP) functionality. The PCI Express Virtual Channel Service Driver allows fully independent flow control between different virtual channels. Advance Error Reporting permits for more robust error reporting than offered by standard baseline mechanism implemented in PCI system. PCI Express Hot-Plug Native Service Driver is responsible for communication between Hot-Plug Controller and operating system. Utilized programming interface is standardized, as presented in Fig. 2, therefore it is not required to modify driver in the case of different hardware architecture. To support several service drivers per device, new architecture of PCI Express Driver Model was implemented. During PCI device enumeration the PCI Express Bus Port Driver is assigned to each PCI Express Port. The Bus Driver analyze set of functionalities supported by port, creates corresponding service devices and register it into system device hierarchy. The Linux kernel represents PCI Express service devices as pseudo-files in *sysfs* file system located in directory */sys/bus/pci_express/devices/*. The name of each device, e.g. *0000:00:1c.0:pcie01*, contains geographical location of the slot, constant string *pcie*, identifier of slot type and identifier of service type. Available identifiers and its meaning are presented in Table I and II. Values in brackets are used by older version of Linux kernel. At the end of PCI Express device enumeration process appropriate service device drivers are loaded.

TABLE I
Available PCI Express port type IDs.

port type	*description*
0	Root Port
1	Upstream Port
2	Downstream Port

TABLE II
Available PCI Express service type IDs.

service type	*description*
1 (0)	Power Management Service
2 (1)	Advanced Error Reporting Service
4 (2)	Native Hot-Plug Service
8 (3)	Virtual Channel Service

Example PCI Express-based system consisting of Root Complex and PCI Express Switch is presented in Fig. 3. Each

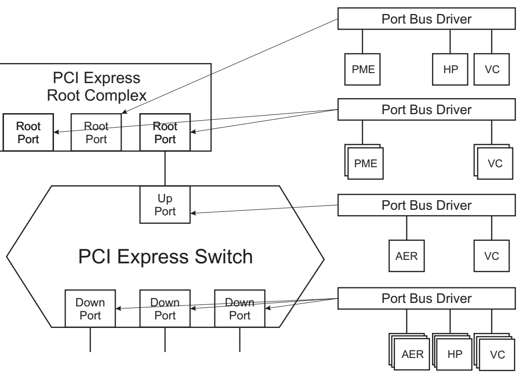


Fig. 3. PCI Express Bus Driver Architecture [6].

PCI Express Port has associated Port Bus Driver with loaded set of service drivers, described in details in Table III. The overview of PCI system visible from bus tree point of view is presented in Fig. 4. Devices *01.0, 1c.0, 1c.3* are PCI-to-PCI Express bridges.

IV. Fake Hot-Plug Driver Functionality

Fake Hot-Plug driver allows emulate remove or attach selected device in a powered up system using software control. Driver does not utilize functionality offered by PCI Express

TABLE III
DESCRIPTION OF PORTS AVAILABLE IN EXAMPLE SYSTEM.

sysfs file name	*service description*
0000:00:01.0:pcie01	PME on first Root Port
0000:00:01.0:pcie08	VC on first Root Port
0000:00:1c.0:pcie01	PME on second Root Port
0000:00:1c.0:pcie04	HP on second Root Port
0000:00:1c.0:pcie08	VC on second Root Port
0000:00:1c.3:pcie01	PME on third Root Port
0000:00:1c.3:pcie08	VC on third Root Port
0000:04:00.0:pcie12	AER on switch upstream port
0000:04:00.0:pcie18	VC on switch upstream port
0000:05:08.0:pcie22	AER on switch first downstream port
0000:05:08.0:pcie24	HP on switch first downstream port
0000:05:08.0:pcie28	VC on switch first downstream port
0000:05:09.0:pcie22	AER on switch second downstream port
0000:05:09.0:pcie24	HP on switch second downstream port
0000:05:09.0:pcie28	VC on switch second downstream port
0000:05:0a.0:pcie22	AER on switch third downstream port
0000:05:0a.0:pcie24	HP on switch third downstream port
0000:05:0a.0:pcie28	VC on switch third downstream port

```
-[0000:00]-+-00.0
           +-01.0-[04-08]----00.0-[05-08]--+-08.0-[08]--
           |                               +-09.0-[07]--
           |                               \-0a.0-[06]--
           +-1b.0
           +-1c.0-[03]----00.0
           +-1c.3-[02]----00.0
           +-1d.0
           +-1d.1
           +-1d.2
           +-1d.3
           +-1e.0-[01]----00.0
           +-1f.0
           +-1f.1
           \-1f.3
```

Fig. 4. The overview of PCI system available in example system.

Hot-Plug system, therefore it is possible to use it on ports that do not support Hot-Plug. Each PCI and PCI Express device is represented by the separate directory in *sysfs* file system e.g. */sys/bus/pci/slots/0000:03:00.0/*. To remove device from the system user has to send value 0 to file called *power* located in directory related to utilized device, e.g.:

```
echo 0 > /sys/bus/pci/slots/0000:03:00.0/power
```

To force bus reenumerate, user has to send value 1 to file called *power* located in directory representing address of the parent or sibling device, e.g.:

```
echo 1 > /sys/bus/pci/slots/0000:00:1c.0/power
```

Internally each PCI and PCI Express device is represented by structure located in linked list. During initialization driver register notification handler that allows to correctly serve standard Hot-Plug events like device remove or device add, therefore list of available devices is always up-to-date. Presented functionality of fake Hot-Plug driver is available starting from kernel version 2.6.30.

V. HOT-PLUG FOR FPGA-BASED PCI EXPRESS ENDPOINTS

Reprogramming of FPGA-based PCI Express end-point device does not emit standard Hot-Plug event and in consequence cannot be correctly recognized by Hot-Plug Controller. After reprogramming internal structures of operating system storing information about device configuration and capabilities are invalid, in consequence every access to hardware component will lead to kernel exception. To restore full functionality, operating system must reenumerate part of the bus and once again perform resource allocation. This operation can be performed by fake Hot-Plug driver but must be initiated manually be the user. It is important to note that correct support for reprogrammable devices is available only in system with implemented new version of Hot-Plug Native Service Driver.

During the development of FPGA-based devices, it is very important to correctly manage versions of currently tested firmware. Wrong software downloaded to device can result in unpredictable behavior of the system and lead to problem with machine and operators safety. Based on functionality offered by *udev* device manager it is possible to check correctness of used firmware during Hot-Plug event processing. The *udev* is a device manager utilized by Linux kernel 2.6. It is respon-

```
ACTION=="add", SUBSYSTEM=="pci", KERNEL=="0000:03:00.0",
ATTR{vendor}=="0x10EE", ATTR{device}=="0x0008",
RUN+="/bin/bash /etc/develop/check_firmware 0x10EE0008"

ACTION=="add", SUBSYSTEM=="pci", KERNEL=="0000:02:00.0",
ATTR{vendor}=="0x10EE", ATTR{device}=="0x009",
RUN+="/bin/bash /etc/develop/check_firmware 0x10EE0018"
```

Fig. 5. Configuration file that describe behavior of *udev* device manager when device with device ID equal to 0x0008 or 0x0009 appears in specified PCI Express slots in the system.

sible for dynamic allocation of device files in */dev* directory, implementation of persistent device naming convention and firmware load. It works as a system daemon and process events sent out by kernel through *netlink* interprocess communication mechanism [7]. Behavior of *udev* manager is controlled by the set of configuration files located in directories */etc/udev/ruses.d* and */lib/udev/rules.d* containing rules describing initialization of devices identified by the set of parameters. Example of such file is presented in Fig. 5. Both rules are applied when specified device is connected to the system. Every parameters related to the specified device, that can be used to create *udev* rules, can be viewed by tool called *udevadm*, see Fig.6.

```
udevadm info --path=/sys/bus/pci/devices/0000\:03\:00.0/
--attribute-walk

looking at device '/devices/pci0000:00/0000:00:1c.0/
0000:03:00.0':

    KERNEL=="0000:03:00.0"
    SUBSYSTEM=="pci"
    DRIVER==""
    ATTR{vendor}=="0x10ee"
    ATTR{device}=="0x0008"
    ATTR{subsystem_vendor}=="0x10ee"
    ATTR{subsystem_device}=="0x0007"
    ATTR{class}=="0x050000"
    ATTR{irq}=="255"
    ATTR{local_cpus}=="ffffffff"
    ATTR{local_cpulist}=="0-31"
    ATTR{modalias}=="pci:v000010EEd00000008sv000010EEsd
    00000007bc05sc00i00"
    ATTR{enable}=="0"
    ATTR{broken_parity_status}=="0"
    ATTR{msi_bus}==""
```

Fig. 6. Information about selected PCI Express device acquired by *udevadm* tool.

After Hot-Plug event forced by fake HP driver, *udev* manager based on parameters like geographic address of the device on PCI Express bus (KERNEL parameter), vendor ID and device ID (accordingly ATTR{vendor} and ATTR{device}) selects application to execute (RUN parameter). Script called *check_firmware*, presented in Fig. 7, loads simple PCI Express driver that allows to read raw data from device and generate for each BAR separate file in */dev* directory. Information about version of the firmware is located in BAR zero at offset zero, therefore program *pcie-rw* reads this memory area. Returned value is compared with expected firmware version and in the case of mismatch device is removed from system by fake Hot-Plug driver and appropriate message is logged into the system. At the end of script simple PCI Express driver is removed from the system. System log records generated by

```
#!/bin/bash
cd /etc/develop/

./load_simple_driver

VER_R=$(./pcie-rw /dev/pcie_bar_0 r 0i 1i h)
if [ ${1} != ${VER_R} ]; then
      logger "Wrong firmware version (${VER_R})"
      echo 0 > /sys/bus/pci/slots/0000:00:1c.0/power
else
      logger "Loaded firmware version ${1}"
fi

./unload_simple_driver
cd -
```

Fig. 7. Source code of *check_firmware* script executed by *udev* when new, FPGA-based PCI Express endpoint is discovered.

simple PCI Express driver are presented in Fig. 8. Three PCI Express BARs were discovered, in consequence script *load_simple_driver* creates three devices */dev/pcie_bar_0*, */dev/pcie_bar_1* and */dev/pcie_bar_2*. After firmware checking, driver and related device files were removed from the system.

```
PCIe Bar Read Driver: Init
PCIe Bar Read Driver: New device ID 10EE:0008
pcie_rd_drv 0000:03:00.0: PCI INT A -> GSI 16
                          (level, low) -> IRQ 16
pcie_rd_drv 0000:03:00.0: setting latency timer to 64

BAR 0 start : 0x50003000
BAR 0 end   : 0x500037FF
BAR space   : 2047 bytes (0 MB)
PCIe Bar Read Driver: Bar 0 Device (251, 0)

BAR 1 start : 0x50002000
BAR 1 end   : 0x50002FFF
BAR space   : 4095 bytes (0 MB)
PCIe Bar Read Driver: Bar 1 Device (251, 1)

BAR 2 start : 0x50000000
BAR 2 end   : 0x50001FFF
BAR space   : 8191 bytes (0 MB)
PCIe Bar Read Driver: Bar 2 Device (251, 2)

PCIe Bar Read Driver: Removing device ID 10EE:0008
PCIe Bar Read Driver: PCIe bar 0 released.
PCIe Bar Read Driver: PCIe bar 1 released.
PCIe Bar Read Driver: PCIe bar 2 released.
PCIe Bar Read Driver: Removed device ID 10EE:0008
pcie_rd_drv 0000:03:00.0: PCI INT A disabled
PCIe Bar Read Driver: Release
```

Fig. 8. Information reported by simple PCI Express driver after device discover and remove.

VI. Summary

In this paper, authors present detailed description of Hot-Plug mechanism implementation in kernel of Linux operating system. Additionally, example usage of fake Hot-Plug driver in conjunction with *udev* driver manager utilized to check the correctness of firmware loaded to FPGA-based end-point devices were proposed.

Acknowledgement

The research leading to these results has received funding from the European Commission under the EuCARD FP7 Research Infrastructures grant agreement no. 227579 and Polish National Science Council Grant 642/N-TESLA-XFEL/09/2010/0. The authors are scholarship holders of project entitled ”Innovative education ...” supported by European Social Fund.

References

[1] D. Makowski, G. Jablonski, A. Piotrowski, W. Koprek, W. Cichalewski, W. Jalmuzna, and S. Simrock, “Survey of communication links for ATCA in physics,” *ICALEPCS 2009, October 12-16*, 2009.
[2] T. Kucharski, A. Piotrowski, D. Makowski, and G. Jablonski, “PCIExpress communication layer for ATCA based linear accelerator control system,” *MIXDES 2009 - Mixed Design of Integrated Circuits and Systems*, 2009.
[3] A. H. Wilen, J. P. Schade, and R. Thornburg, *Introduction to PCI Express: A Hardware and Software Developer’s Guide*. Intel Press, 2003.
[4] PICMG, “PCI Express Base Specification 1.1,” PICMG, Tech. Rep., 2005.
[5] R. Budruk, D. Anderson, and E. Solari, *PCI Express System Architecture*. Pearson Education, 2003.
[6] T. L. Nguyen, D. L. Sy, S. Carbonari, and R. Olsson, “Contents PCI Express port bus driver support for Linux,” *Proceedings of the Linux Symposium July 20th 23rd, Ottawa, Canada*, 2005.
[7] G. Kroah-Hartman, “udev A Userspace Implementation of devfs,” *Proceedings of the Linux Symposium July 23th 26rd, Ottawa, Canada*, 2003.

Adam Piotrowski received the MSc and PhD degrees in computer science at the Department of Microelectronic and Computer Science at Technical University of Lodz, in 2004 and 2010 respectively. His research interests include compilation techniques, embedded and fast data acquisition systems. He is involved in development of ATCA-based LLRF control system for XFEL accelerator.

Dariusz Makowski received PhD degree in electrical engineering at the Department of Microelectronics and Computer Science Technical University of Łódź in 2006. His main areas of interests are digital electronics, embedded systems and programmable devices. He is engaged in the development of xTCA standards for High Energy Physics. He is involved in the design of distributed data acquisition and control systems based on ATCA, MTCA and AMC standards.