

PCle Tool to Analysis, Debug and Validate

Submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering / Technology degree in Electronics and Communication Engineering.

By

Vuddagiri Teja Venkata Sai Naveen (38130241)



DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

SCHOOL OF ELECTRONICS AND COMMUNICATION ENGINEERING

SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited with Grade "A" by NAAC | 12B Status by UGC | Approved by AICTE

JEPPIAAR NAGAR, RAJIV GANDHI SALAI, CHENNAI – 600119

MARCH – 2022



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)

Accredited with “A” grade by NAAC | 12B Status by UGC | Approved by AICTE

Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai – 600 119

www.sathyabama.ac.in

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of Vuddagiri Teja Venkata Sai Naveen (38130241) who carried out the project entitled PCIe tool to Analyze, Debug and Validate under my supervision from November – 2021 to April – 2022.

Internal Guide

DR.E.ANNADEVI, M.E., Ph.D.

Head of the Department

DR.T.RAVI, M.E., Ph.D.

Submitted for Viva voce Examination held on

Internal Examiner

External Examiner

DECLARATION

I Vuddagiri Teja Venkata Sai Naveen hereby declare that the Project Report entitled PCIe tool to analyze, debug and validate done by me under the guidance of Dr.E.Annadevi, M.E., Ph.D and Mr. Sundarvinayagam. P, Senior Technical Lead and Mr. Praveen Kumar, Senior Technical Lead at HCL Technologies is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering / Technology degree in Electronics and Communication Engineering.

Date:

Place:



Signature of the Candidate

ACKNOWLEDGEMENT

I am pleased to acknowledge my sincere thanks to Board of Management of SATHYABAMA for their kind encouragement in doing this project and for completing it successfully. I am grateful to them.

I convey my thanks to Dr.N.M.Nanditha., Dean, School of Electronics and Dr.T.Ravi, Head of the Department, Dept. of Electronics for providing me necessary support and details at the right time during the progressive reviews.

I would like to express my sincere and deep sense of gratitude to my Project Guide Dr.E.Annadevi Mam for her valuable guidance, suggestions and constant encouragement paved way for the successful completion of my project work.

I wish to express my thanks to all Teaching and Non-teaching staff members of the Department of Electronics who were helpful in many ways for the completion of the project.

ABSTRACT

We are building a tool which have a capacity to control all the PCIe devices in a system. This tool has an add-on capabilities of analyzing devices present in system like the lspci command does and debug the devices based on the configuration and check for errors and signals that represent the working condition of the device. Finally, we have a feature of validate the device, in this we validate the device connection and legitimate of the device and overall performance measurements.

PCIe Abbreviated as Peripheral Component Interconnect express. PCIe is a motherboard expansion bus standard introduced in 2003 to enable high speed serial communication between the CPU and its peripheral components. It is the standard and most popular communication method at present.

The history of PCIe is started in 90s and named as PCI (Peripheral Component Interface). It is used as the component bus with other standard buses for data transfer, soon it is evolved into greater heights with high data transfers. The PCIe is started in 2003 before that PCI has data transfer speed of 133MB/s and high version has 1GB/s. Then PCIe started with 2.5GB/s next 5GB/s and goes on increasing till now. At present PCIe bus is capable of 64GB/s data transfer rate which is extremely high and fastest of all.

PCIe communication is hierarchical. There is a single source through which all the data passes which is the root complex. The information passes to the root complex from multiple PCIe endpoints. Link is composed of lanes; these are nothing but number of channels present to communicate as if the PCIe device is x8 then it has 8 lanes to transfer. Lane consists of differential sequential pairs with one pair for receiving data and the other for transmitting. Thus each lane composed of 4 wires.

The device driver searches for the PCIe system starting at bus 0 and locates all PCIe devices and bridges in the system. It builds a topology of the system using linked lists.

CONTENTS

S.No	TOPIC	Page No
1)	Chapter-1: Introduction	1
	1.1 - PCIe Overview	1
	1.2 - PCIe Topology	5
	1.3 - BDF Number	7
	1.4 - PCIe Enumeration	8
	1.5 - Configuration Space	10
	1.6 - Header Types	11
2)	Chapter-2: Literature Survey	19
3)	Chapter-3: Aim and Scope	20
	3.1 - Project Uses	20
	3.2 - Goals of the Project	20
	3.3 - Dependencies Needed	20
	3.4 - Milestones	20
4)	Chapter-4: Code	
	4.1 – LSPCI Code	21
	4.2 – Code to read PCI Device	23
	4.3 - Code to read config space	24
5)	Chapter-5: Results and Analysis	33
6)	Reference	36

List of Abbreviations

- 1) PCI – Peripheral Component Interconnect
- 2) PCIe – Peripheral Component Interconnect Express
- 3) lspci – Command in Ubuntu Linux
- 4) BDF – Bus, Device, Function Number
- 5) NIC – Network Interface Card
- 6) DMA – Direct Memory Access
- 7) SoC – System on Chip

List of Figures

- 1) Figure 1.1 – PCIe Terminology
- 2) Figure 1.2 – PCIe Connection to CPU
- 3) Figure 1.3 – Real life PCIe slot
- 4) Figure 1.4 – PCIe Topology
- 5) Figure 1.5 – PCIe Bar Bits
- 6) Figure 1.6 – Header
- 7) Figure 1.7 – Type 0 Header
- 8) Figure 1.8 – Type 1 Header
- 9) Figure 1.9 – Capability Linked List
- 10) Figure 5.1 – Lspci my code execution
- 11) Figure 5.2 – Standard lspci command execution
- 12) Figure 5.3 – Code execution of IO mapped code
- 13) Figure 5.4 – Code execution of IO mapped code
- 14) Figure 5.5 – Code execution of read config space

CHAPTER-1: INTRODUCTION

1.1 OVERVIEW:

Peripheral Component Interconnect Express (PCIe or PCI-E) is a serial expansion bus standard for connecting a computer to one or more peripheral devices.

PCIe provides lower latency and higher data transfer rates than parallel busses such as PCI and PCI-X. Every device that's connected to a motherboard with a PCIe link has its own dedicated point-to-point connection. This means that devices are not competing for bandwidth because they are not sharing the same bus.

Peripheral devices that use PCIe for data transfer include graphics adapter cards, network interface cards (NICs), storage accelerator devices and other high-performance peripherals.

With PCIe, data is transferred over two signal pairs: two wires for transmitting and two wires for receiving. Each set of signal pairs is called a "lane," and each lane is capable of sending and receiving eight-bit data packets simultaneously between two points.

PCIe Communication:

- 1) Point to Point
- 2) Bi-directional
- 3) Backwards Compatible
- 4) Widely Adopted

1) Point to Point: Point to point simply means there is one participant at each end like a phone call. Let's say we have 2 devices, device A and device B.

If device A wants to communicate with device B, it will do so by PCIe data link.

- 2) **Bi-directional:** It means that device A and device B can transmit and receive the data from both sides.
- 3) **Backwards Compatible:** Each PCIe version is backwards compatible to its previous versions. When end point contained devices that are from different PCIe versions the lanes automatically functions at the slower data rate of the earlier PCIe version.
- 4) **Widely Adopted:** PCIe applications have expanded far beyond the original PC market to include.

Ex: Data centers, storage systems, servers, vehicle based systems etc.

PCIe Features:

- 1) Dual simplex point to point serial communication
- 2) Independent transmit and receive sides
- 3) Scalable link widths
- 4) x1, x2, x4, x8, x12, x16, x32.
- 5) Scalable link speeds
- 6) 2.5, 5.0 and 8.0 GT/s
- 7) Packet based transaction protocol

PCIe Terminology:

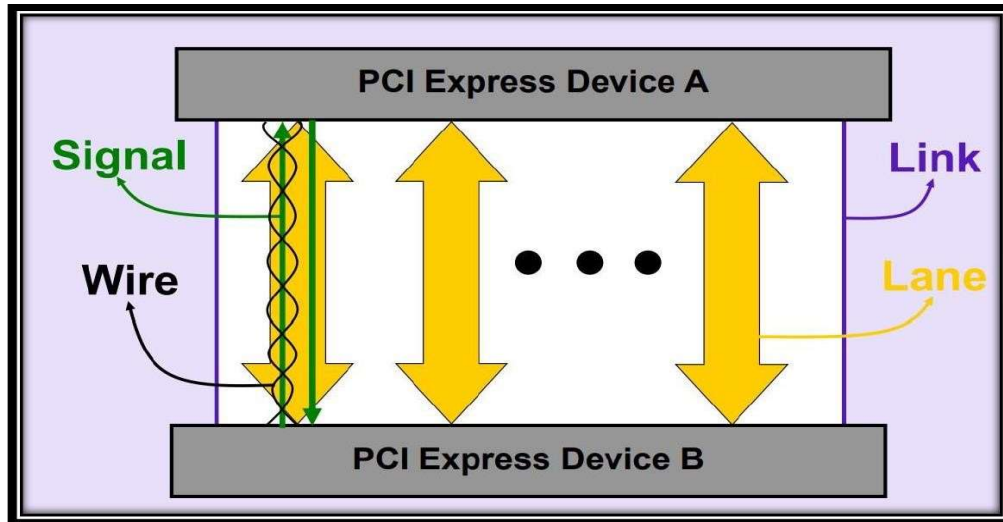


Figure 1.1 – PCIe terminology

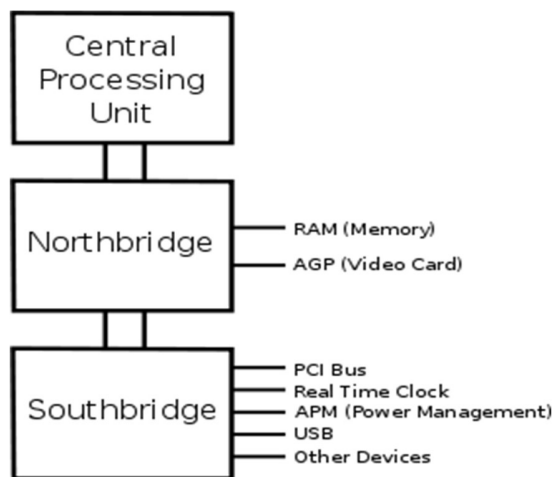


Figure 1.2 – PCIe connection to CPU

As we can see in the above image that PCIe bus is connected to south bridge. These bridges in the motherboard is core logic chipset. South bridge typically implements the slower capabilities of the motherboard in the chipset. As in above

image, only north bridge is connected directly to CPU and it establishes the connection between CPU and RAM directly to obtain faster memory access.

To start a PCIe device there are certain steps to follow. First the link should be established, the link is established when we plug the PCIe device into motherboard based on the configuration this is x4 or x8 it will establish a connection and that connection establishment is called link. Link connection consists of lanes and these lanes consists of differential pair of transmitter and receiver to establish a full duplex connection. This pair of transmitter and receiver makes a lane and group of lanes is called link. The number of lanes is calculated by x4 or x8 like x4 makes 4 lanes and x8 makes 8 lanes.

In previous we have seen how to establish a connection of PCIe devices and now we see about how to start the PCIe device it needs a REFCLK it's a Reference Clock that triggers the PCIe device to start the data transfer through link and lane.

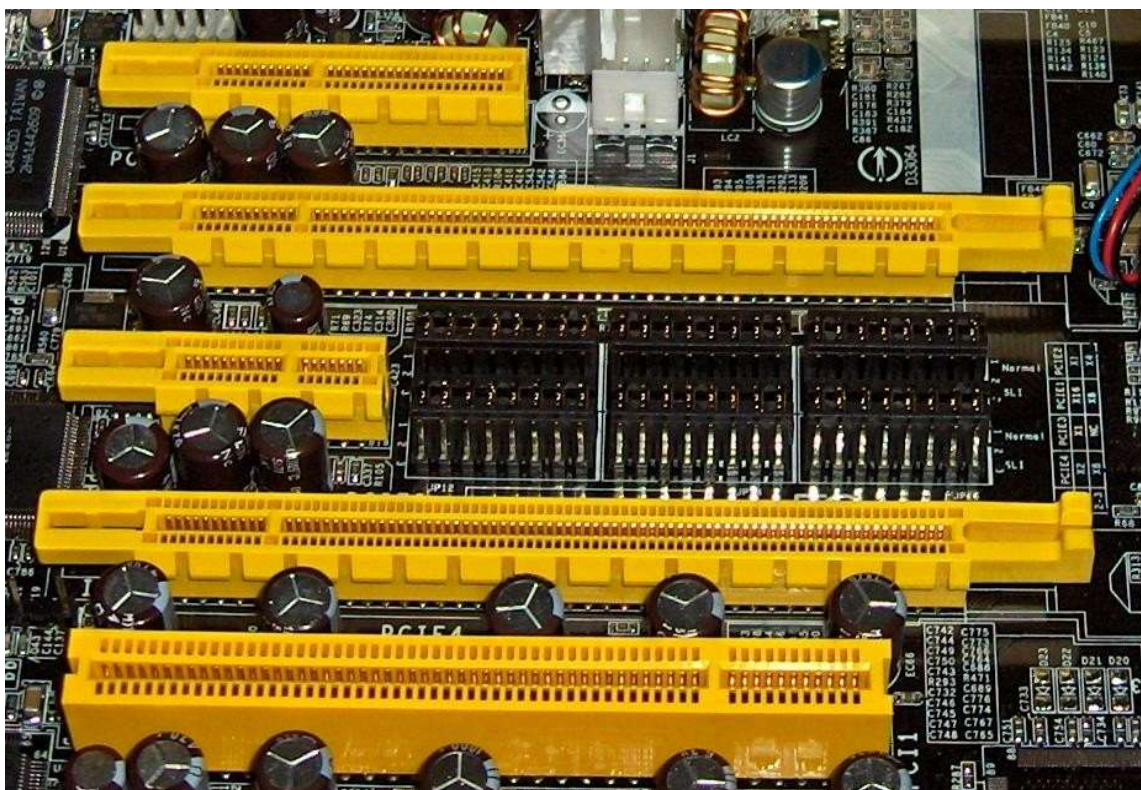


Figure 1.3 – Real life PCIe slot

1.2 PCIe Topology:

A PCIe device tree consists of root-complex, endpoints, switches and bridges etc.

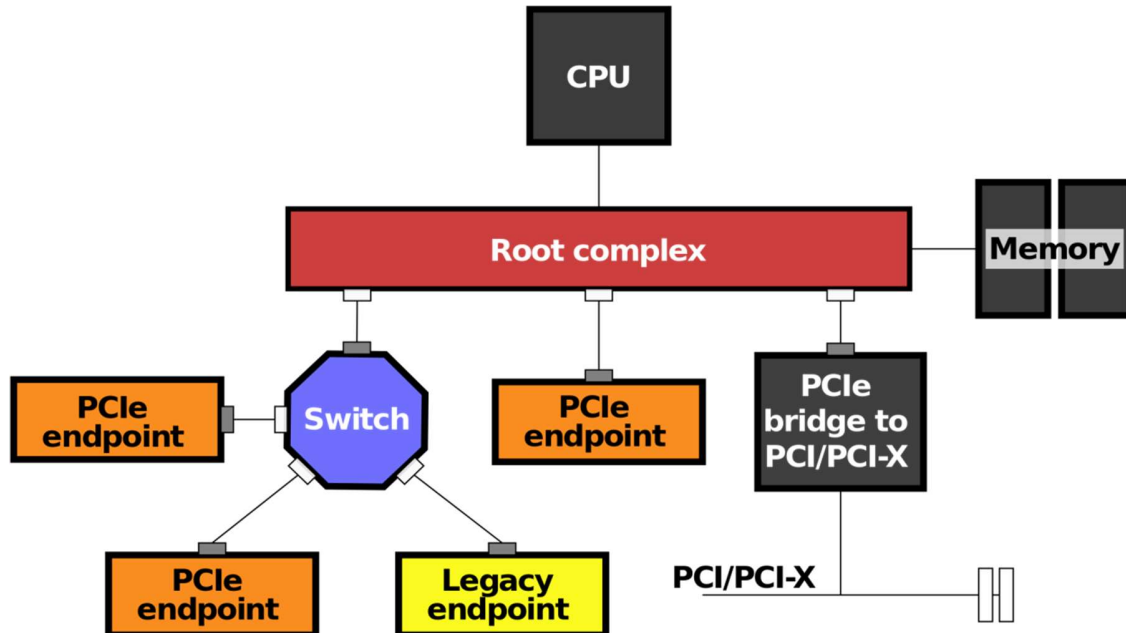


Figure 1.4 – PCIe Topology

Root-Complex:

A Root-Complex connects the all PCIe devices to CPU and memory. It's like a management unit of PCIe devices. It has a subsystems of switches, repeaters, bridges to connect PCIe devices of all types in ports of root-complex.

The PCIE Root Complex holds a master copy of a 'Type 1 Configuration Table' that defines the host memory space that is accessible from each Endpoint device. In addition, each PCIE Endpoint device holds a master copy of their own memory space map in the host system memory as a 'Type 0 Configuration Table', this configuration table in each device allows the host to access the local memory of a PCIe device. Both the Type 1 and Type 0 configuration tables are set up by the Host Operating System that controls the Root Complex by a process known as enumeration and

which acts to build a device memory map for the system by querying each bridge, and endpoint device connected on the bus network.

Switches:

A PCIe switch is an expansion of the PCIe hierarchy. It's like a plug box in our home the operation is similar to the plug box concept as plug box supply power from one socket to all the sockets present in the box equally. The switch also does the same by one upstream which is connected to root-complex and multiple downstream ports and switching logic that routes TLPs between ports and downstream ports are connected to PCIe endpoints and those will be discussed later in this section.

Switches connected multiple PCIe devices to the root-complex and it is called upstream and the PCIe devices are endpoints to the switches. A PCIe switch is transparent means that software and other devices have nothing to do with the presence of this device and no driver is requiring to operate this switch.

Bridges:

PCIe is the most used bus standard in the modern devices, motherboards and expansion cards. But still some devices support legacy PCI ports to support backward compatibility and bridges give the way to those backward compatible devices to communicate to the root-complex and work normally as they do. This is the main difference between the switches and bridges in PCIe hierarchy.

PCIe bridge support both forward and reverse modes as PCI device connected through bridge can communicated in both ways like half duplex communication but it is purely based on the primary and secondary side on which device takes on. This can't be possible in switches. the PCIe bridge handles the interface conversion effectively with minimal impact to the PCI or PCIe hardware.

In Forward mode, the PCIe bridge connects a PCIe host device, which takes the primary side, to the PCI interface and ultimately to the PCI device. Some bridges also support fan-out bridging where several PCI devices can be connected to the PCIe bridge to work with the PCIe host device. In this mode, the PCIe host device

handles the PCIe bridge as the PCIe endpoint while the PCI device handles the PCIe bridge as the PCI host device.

In Reverse mode, the PCI host device takes on the primary side and connects to the PCIe bridge which is linked to the PCIe endpoint devices. The PCI host device handles the PCIe bridge as the PCI endpoint while the PCIe device handles the bridge as the PCIe host. This is mostly used to connect PCI CPUs to PCIe devices.

Endpoints:

Endpoints are peripheral devices like Ethernet, USB or GPU. Endpoints initiate transactions as a requester or respond to transactions as a completer. There are two types of endpoints, PCIe endpoints and legacy endpoints. These legacy endpoints support I/O transactions, and may support locked transaction semantics as a completer but not as a requester.

Legacy endpoints required to support 64-bit memory- addressing capability. PCIe endpoints must not support I/O or locked transaction semantics and must support MSI style interrupt generation.

1.3 BDF Number:

BDF stands for the Bus, Device, Function notation used to succinctly describe PCI and PCIe devices. The simple form of the notation is:

1.3.1 PCI Bus number in hexadecimal, often padded using a leading zero to two or four digits

1.3.2 A colon (:)

1.3.3 PCI Device number in hexadecimal, often padded using a leading zero to two digits. Sometimes this is also referred to as the slot number.

1.3.4 A decimal point (.)

1.3.5 PCI Function number in hexadecimal

For example, the following describes Bus 0, Device 2, Function 0:

00:02.0

BDF Notation Extension for PCI Domain:

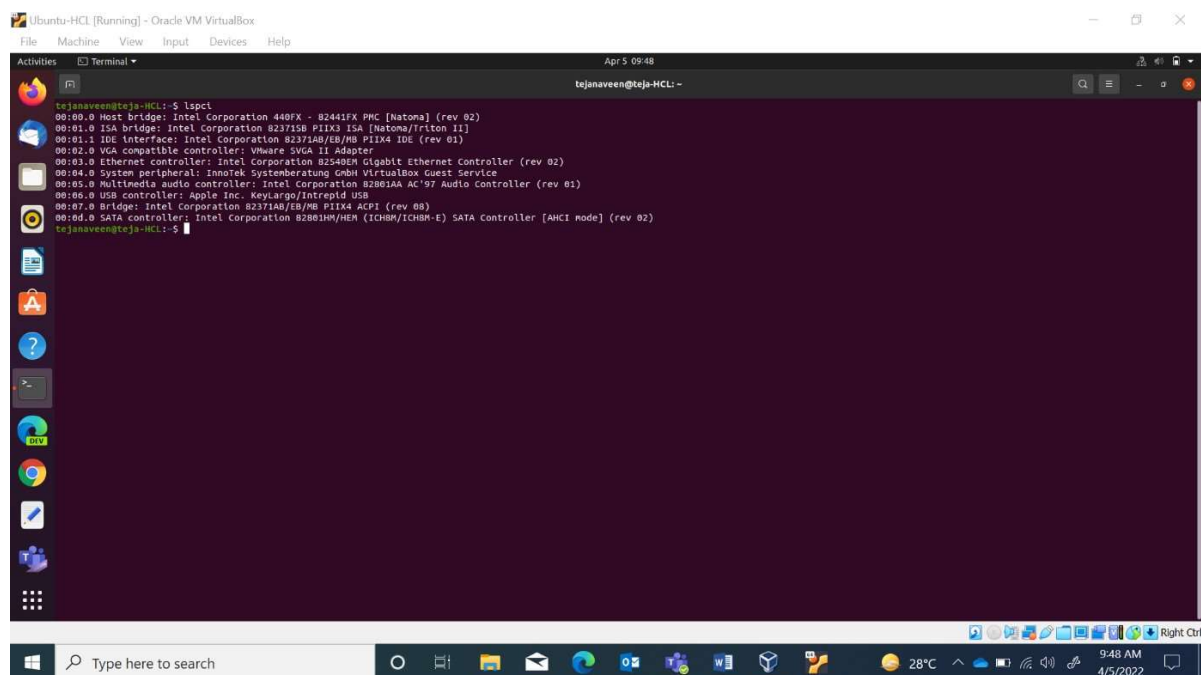
There is an extended form of the BDF notation, which confusingly is also typically referred to as BDF. Fortunately, both variants are typically interchangeable.

The extension of the notation is to prefix the simple notation with:

- 1) PCI Domain number, often padded using leading zeros to four digits
- 2) A colon (:)

For example, the following describes Domain 0, Bus 0, Device 2, Function 0:

0000:00:02.0



```
tejanaveen@teja-HCL:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: VMware SVGA II Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet controller (rev 02)
00:04.0 System peripheral: Intel Corporation 82801AA AC'97 Audio controller (rev 01)
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio controller (rev 01)
00:06.0 USB controller: Apple Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801JB/MB/7MB (ICH8/ICH8M-E) SATA controller [AHCI mode] (rev 02)
tejanaveen@teja-HCL:~$
```

1.4 PCIe Enumeration:

PCI Express (PCIe) utilizes a point to point interconnect and uses switches to fan out and expand the number of PCIe connections in a system. Upon system boot up a critical task is the discovery or enumeration process of all the devices in the PCIe tree so they can be allocated by the system software.

During the enumeration process the system software discovers all of the switch and endpoint devices that are connected to the system, determines the memory requirements and then configures the PCIe devices. The PCIe switch devices represent a special case in this process as their configuration is unique and separate from that of PCIe endpoints.

In the simulation test bench environment however, only their configuration is required; the discovery process is not strictly necessary as the number of PCIe devices are known ahead of time. This paper will elucidate the process of switch configuration using Synopsys' PCI Express VMM simulation class libraries for System- Verilog.

First, while the discovery process is not needed within the testbench environment, the test bench must still select the bus numbers and memory address of all the devices. These bus numbers are shown in Figure 1.4 as Bus#2, Bus#3, Bus#4, and Bus#5. Figure 1.4 shows the testbench that is used in the remainder of the article.

Now that the bus numbers, device numbers and address range are known, the Switch can be configured. Each port of the Switch has its own configuration space and the format of the configuration space is type CONFIG 1.

There are a minimum of 6 registers that need to be programmed for the Switch to support configuration and memory accesses to an EP device.

They are:

1. Command : Enable memory access
2. Primary Bus : Bus number of upstream bus
3. Secondary Bus : Bus number of bus directly attached
4. Subordinate Bus : Bus number of farthest downstream

- 5. Memory Limit : Maximum memory address
- 6. Memory Base : Starting memory address

In summary, PCIe designs must go through the process of Switch enumeration to discover available switches, and then to configure them.

1.5 Configuration Space:

Every PCI device in the system, including the PCI-PCI bridges has a configuration data structure that is somewhere in the PCI configuration address space. The PCI Configuration header allows the system to identify and control the device. Exactly where the header is in the PCI Configuration address space depends on where in the PCI topology that device is. For example, a PCI video card plugged into one PCI slot on the PC motherboard will have its configuration header at one location and if it is plugged into another PCI slot then its header will appear in another location in PCI Configuration memory. This does not matter, for wherever the PCI devices and bridges are the system will find and configure them using the status and configuration registers in their configuration headers.

Typically, systems are designed so that every PCI slot has its PCI Configuration Header in an offset that is related to its slot on the board. So, for example, the first slot on the board might have its PCI Configuration at offset 0 and the second slot at offset 256 (all headers are the same length, 256 bytes) and so on.

A system specific hardware mechanism is defined so that the PCI configuration code can attempt to examine all possible PCI Configuration Headers for a given PCI bus and know which devices are present and which devices are absent simply by trying to read one of the fields in the header (usually the *Vendor Identification* field) and getting some sort of error. The describes one possible error message as returning *0xFFFFFFFF* when attempting to read the *Vendor Identification* and *Device Identification* fields for an empty PCI slot.

1.6 Headers Types:

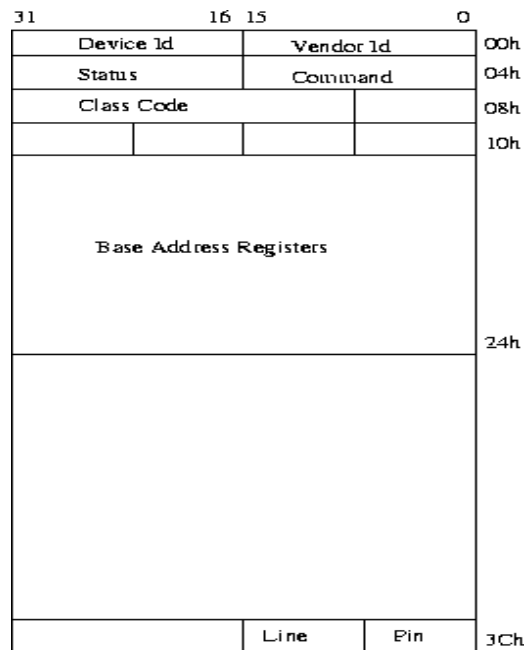


Figure 1.5 – Standard Header

Vendor Identification:

A unique number describing the originator of the PCI device. Digital's PCI Vendor Identification is *0x1011* and Intel's is *0x8086*.

```
pci_read_byte(struct device, PCI_VENDOR_ID);
```

Device Identification:

A unique number describing the device itself. For example, Digital's 21141 fast Ethernet device has a device identification of *0x0009*.

```
pci_read_byte(struct device, PCI_DEVICE_ID);
```

Status Register:

This field gives the status of the device with the meaning of the bits of this field set by the standard. Simply states the error and device information.

```
pci_read_byte(struct device, PCI_STATUS);
```

Command Register:

By writing to this field the system controls the device, for example allowing the device to access PCI I/O memory.

```
pci_read_byte(struct device, PCI_COMMAND);
```

Class Code:

This identifies the type of device that this is. There are standard classes for every sort of device; video, SCSI and so on. The class code for SCSI is *0x0100*.

We can retrieve this class code from configuration space using the below API.

```
pci_read_byte(struct device, PCI_CLASS_DEVICE);
```

Base Address Registers

Base Address Registers (or BARs) can be used to hold memory addresses used by the device, or offsets for port addresses. Typically, memory address BARs need to be located in physical ram while I/O space BARs can reside at any memory address (even beyond physical memory). To distinguish between them, you can check the value of the lowest bit. The following tables describe the two types of BARs:

PCI BAR Bits		
Bits	Description	Values
<i>For all PCI BARs</i>		
0	Region Type	0 = Memory 1 = I/O
<i>For Memory BARs</i>		
2-1	Locatable	0 = any 32-bit 1 = < 1 MB 2 = any 64-bit
3	Prefetchable	0 = no 1 = yes
31-4	Base Address	naturally 16-byte aligned
<i>For I/O BARs</i>		
1	Reserved	
31-2	Base Address	naturally 4-byte aligned

Figure 1.6 – BAR Bits

The Type field of the Memory Space BAR Layout specifies the size of the base register and where in memory it can be mapped. If it has a value of 0x0 then the base register is 32-bits wide and can be mapped anywhere in the 32-bit Memory Space. A value of 0x2 means the base register is 64-bits wide and can be mapped anywhere in the 64-bit Memory Space (A 64-bit base address register consumes 2 of the base address registers available). A value of 0x1 is reserved as of revision 3.0 of the PCI Local Bus Specification. In earlier versions it was used to support memory space below 1MB (16-bit wide base register that can be mapped anywhere in the 16-bit Memory Space).

When a base address register is marked as Prefetchable, it means that the region does not have read side effects (reading from that memory range doesn't change any state), and it is allowed for the CPU to cache loads from that memory region and read it in bursts (typically cache line sized). Hardware is also allowed to merge repeated stores to the same address into one store of the latest value. If you are using paging and want maximum performance, you should map prefetchable MMIO regions as WT (write-through) instead of UC (uncacheable). On x86, frame buffers are the exception, they should be almost always be mapped WC (write-combining).

Address and size of the BAR

When you want to retrieve the actual base address of a BAR, be sure to mask the lower bits. For 16-bit Memory Space BARs, you calculate $(\text{BAR}[x] \& 0\text{xFFF0})$. For 32-bit Memory Space BARs, you calculate $(\text{BAR}[x] \& 0\text{xFFFFFFF0})$. For 64-bit Memory Space BARs, you calculate $((\text{BAR}[x] \& 0\text{xFFFFFFFF0}) + ((\text{BAR}[x + 1] \& 0\text{xFFFFFFFF}) \ll 32))$. For I/O Space BARs, you calculate $(\text{BAR}[x] \& 0\text{xFFFFFFF0})$.

To determine the amount of address space needed by a PCI device, you must save the original value of the BAR, write a value of all 1's to the register, then read it back. The amount of memory can then be determined by masking the information bits, performing a bitwise NOT ('~' in C), and incrementing the value by 1. The original value of the BAR should then be restored. The BAR register is naturally aligned and as such you can only modify the bits that are set. For example, if a device utilizes 16 MB it will have BAR0 filled with 0xFF000000 (0x1000000 after decoding) and you can only modify the upper 8-bits.

Interrupt Pin:

Four of the physical pins on the PCI card carry interrupts from the card to the PCI bus. The standard labels these as A, B, C and D. The *Interrupt Pin* field describes which of these pins this PCI device uses. Generally, it is hardwired for a particular device. That is, every time the system boots, the device uses the same interrupt pin. This information allows the interrupt handling subsystem to manage interrupts from this device.

```
pci_read_byte(struct device, PCI_INTERRUPT_PIN);
```

Interrupt Line:

The *Interrupt Line* field of the device's PCI Configuration header is used to pass an interrupt handle between the PCI initialization code, the device's driver and Linux's interrupt handling subsystem. The number written there is meaningless to the device driver but it allows the interrupt handler to correctly route an interrupt from the PCI device to the correct device driver's interrupt handling code within the Linux operating system.

```
pci_read_byte(struct device, PCI_INTERRUPT_LINE);
```

PCIe Configuration Request:

The request will be sent to all devices based on the type of headers like whether they are type 0 or type 1 header configured and further to the endpoints I mean at starting the root complex sends type 0 request to endpoints and type 1 request to switches/bridges and further those bridges also have endpoints so those bridges convert the request from type 1 to type 0 automatically.

Type 0 Header:

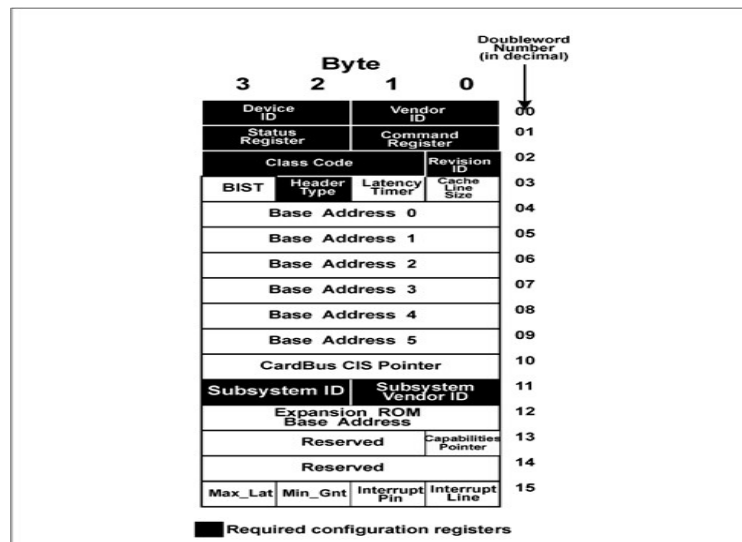


Figure 1.7 – Type 0 Header

Revision ID:

Gives the revised identifier of a device which stores the valid ID provided by the vendor.

```
pci_read_byte(struct device, PCI_REVISION_ID);
```

BIST Register:

BIST means Built in Self-Test register and it is a status register which authorizes the control of device.

Header Type:

This register represents the device type whether it is type 0 or type 1 and it also mentions the device is multi-functional or not.

Latency Timer:

Measures latency in units of PCI bus clocks.

Cache Line Size:

Specifies the system cache size in 32 bit units and it has garbage value then the device behaves as 0 stored in it.

In order to allow more parts of configuration space to be standardized without conflicting with existing uses, there can be a list of *capabilities* defined within the remaining 192 bytes of PCI configuration space. Each capability has one byte that describes which capability it is, and one byte to point to the next capability. The number of additional bytes depends on the capability ID. If capabilities are being used, a bit in the *Status* register is set, and a pointer to the first in a linked list of capabilities is provided in the *Cap. pointer* register defined in the Standardized Registers. When a capability register set is enabled it is tied together by a linked list starting with an 8-bit pointer at address 34h in the configuration space header.

Cardbus CIS Pointer:

Points to the card information structure and is used by devices that share silicon between cardbus and PCI.

Type 1 Header:

	31	24	23	16	15	8	7	0
0x0000	Device ID					Vendor ID		
0x0004	Status					Command		
0x0008	Class Code						Revision ID	
0x000C	BIST		Header Type		Primary Latency Timer		Cache Line Size	
0x0010	BAR Registers							
0x0014	BAR Registers							
0x0018	Secondary Latency Timer		Subordinate Bus Number		Secondary Bus Number		Primary Bus Number	
0x001C	Secondary Status				I/O Limit		I/O Base	
0x0020	Memory Limit				Memory Base			
0x0024	Prefetchable Memory Limit				Prefetchable Memory Base			
0x0028	Prefetchable Base Upper 32 Bits							
0x002C	Prefetchable Limit Upper 32 Bits							
0x0030	I/O Limit Upper 16 Bits				I/O Base Upper 16 Bits			
0x0034	Reserved						Capabilities Pointer	
0x0038	Expansion ROM Base Address							
0x003C	Bridge Control				Interrupt Pin		Interrupt Line	

Figure 1.8 – Type 1 Header

Header & Capability Register Sets

- Each capability structure set also has a unique Capability ID
- To find the next capability pointer we look at the base address of the capability register set and add an offset of 01h
- When you reach a capability structure with an ID of 10h then you have reached the PCI Express Capability Structure
- When you reach a capability structure with Next Capability Pointer of 00h then you have reached the end of the linked list

Capability IDs:

```
#define PCI_CAP_LIST_ID      0      /* Capability ID */

#define PCI_CAP_ID_PM      0x01  /* Power Management */

#define PCI_CAP_ID_AGP      0x02  /* Accelerated Graphics Port */

#define PCI_CAP_ID_VPD      0x03  /* Vital Product Data */

#define PCI_CAP_ID_SLOTID    0x04  /* Slot Identification */

#define PCI_CAP_ID_MSI      0x05  /* Message Signalled Interrupts */

#define PCI_CAP_ID_CHSWP     0x06  /* CompactPCI HotSwap */

#define PCI_CAP_ID_PCIX 0x07  /* PCI-X */

#define PCI_CAP_ID_HT      0x08  /* HyperTransport */

#define PCI_CAP_ID_VNDR 0x09  /* Vendor-Specific */

#define PCI_CAP_ID_DBG      0x0A  /* Debug port */

#define PCI_CAP_ID_CCRC     0x0B  /* CompactPCI Central Resource Control */

#define PCI_CAP_ID_SHPC 0x0C  /* PCI Standard Hot-Plug Controller */
```

```

#define PCI_CAP_ID_SSVID      0x0D  /* Bridge subsystem vendor/device ID */

#define PCI_CAP_ID_AGP3 0x0E  /* AGP Target PCI-PCI bridge */

#define PCI_CAP_ID_SECDEV    0x0F  /* Secure Device */

#define PCI_CAP_ID_EXP       0x10  /* PCI Express */

#define PCI_CAP_ID_MSIX 0x11  /* MSI-X */

#define PCI_CAP_ID_SATA 0x12  /* SATA Data/Index Conf. */

#define PCI_CAP_ID_AF        0x13  /* PCI Advanced Features */

#define PCI_CAP_ID_EA        0x14  /* PCI Enhanced Allocation */

#define PCI_CAP_ID_MAX      PCI_CAP_ID_EA

#define PCI_CAP_LIST_NEXT    1      /* Next capability in the list */

#define PCI_CAP_FLAGS        2      /* Capability defined flags (16 bits) */

#define PCI_CAP_SIZEOF      4

```

0	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000	72	11	01	E0	07	01	10	00	01	00	00	FF	10	00	00	00
010	00	00	00	D0	00	00	FC	CF	00	00	F8	CF	00	00	00	00
020	00	00	00	00	00	00	00	00	00	00	00	00	B1	C0	11	18
030	00	00	00	00	50	00	00	00	00	00	00	00	00	0F	01	00
040	00	00	00	00	60	61	00	02	00	00	00	00	00	00	00	00
050	05	78	84	00	00	00	00	00	00	00	00	00	00	00	00	00
060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
070	00	00	00	00	00	00	00	00	01	80	03	00	08	00	00	00
080	10	00	02	00	01	80	64	00	10	28	00	00	81	F4	03	01
090	00	00	81	10	00	00	04	00	00	00	00	00	00	00	00	00

Fig 1.9 Capability linked list

Chapter-2: Literature Survey

Author	Title of the project	Problem Addressed
L. Rota, M. Caselle, S. Chilingaryan, A. Kopmann, M. Weber(2019)	A new DMA PCIe Architecture for Gigabyte Data Transmission	As growing number of scientific experiments on data acquisition of real-time the system must be capable of high data rates over long observation. This can be achieved by using DMA engine to overcome limitations by CPU Scheduling.
Pramod Kumar Tanwar (2019)	Zynq SoC Based high speed data transfer using PCIe: A device driver based approach	Image and Video processing has revolutionized the world with high performance digital cameras. PCIe is used here to setup a link between CPU and FPGA Board.
Kamlendra Chandra, Amit Prakash Jagtap (2019)	Design of PCIe-DMA bridge interface for high speed Ethernet applications	The intensive demand for computational processing has led to development of various accelerators where bridge is served by FPGA connected over PCIe, utilizing potentials of extended Board.
Linsong Gu, Li Li (2020)	Research on PCIe Bus Communication Based on Neokylin	Along with the rapid development of the information technology, higher transmission rate of data signaling is increasingly required by computing systems. Therefore PCIe bus is acknowledged as the representative of the next generation bus standard.

CHAPTER-3: AIM AND SCOPE

3.1 Project Uses:

- 1) With this tool we can easily debug and analyze any PCI/PCIe device connected to the system.
- 2) Scan configuration space for all PCI-visible functions in system.
- 3) Detecting error in PCI devices.
- 4) Read and Write to any config space location, memory address or IO address.
- 5) Create decode files for structures in config space, memory address space and IO space.
- 6) Extended capability structure access.

This project is A HCL owned tool for analyzing PCIe device and its development.

3.2 Goals of the Project:

- 3.2.1 PCIe tool to access config space.
- 3.2.2 Changing and debugging the PCIe Device characteristics
- 3.2.3 PCIe debug tool for validation, analysis and learning that allows the user to read and write any memory, IO or configuration space address.

3.3 Dependencies Needed:

Linux-ubuntu 18.4 or higher, lspci/pciutils

3.4 Milestones:

- 3.4.1 PCIe Architecture
- 3.4.2 Operating on PCIe devices in Linux
- 3.4.3 Basic tool to access the PCIe space.
- 3.4.4 Addition of functionalities to update / monitor the PCIe devices.

CHAPTER-4: CODE

4.1 LSPCI Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/io.h>
```

Initialize all the headers files required for the program such as standard libraries, unified libraries for linux, system i/o related libraries to access the i/o ports.

For easy access define the word or typecast the datatype as WORD, DWORD and define all the wanted fixed values in macros for faster compile and run speed of the program.

```
WORD bus, dev, fun;
```

The above statement makes the variables bus, dev, fun as word sized variables.

```
DWORD addr, data;
```

The above statement makes the variables config data and config address as dword sized variables.

```
int ret;
```

Use a variable to get the values from the kernel since to retrieve a value from a kernel we need a higher privileged variable permissions and that is done using iopl(3) gives the variable a io privilege to the variable and helps to retrieve data from kernel space.

After completion of program remove the privilege to original to remove dangling pointer like in dynamic memory allocation.

```
for (bus = 0; bus <= PCI_MAX_BUS; bus++)
```

By making a loop like above for all bus, device and function and getting address like below statement can retrieve the information from config space of the device from its memory.

```
addr = PCI_BASE_ADDR | (bus << 16) | (dev << 11) | (fun << 8);
```

After making a address to retrieve information from config space we need to retrieve that info from the address we have created and that is done by below statement.

```
outl(addr, CONFIG_ADDR);
```

and by storing it and comparing it gives us the details of BDF numbers of available devices in our system.

```
data = inl(CONFIG_DATA);
```

As I said before we have to make the variable back to original permissions and that is done using the below.

```
ret = iopl(0);
```

This code is similar to the functioning of lspci command source code as this lspci displays all the PCIe devices in a system with BDF number. This code is a simple alternative for the source code and through this we can manipulate the code and create a command that can do more.

We extend this code further to make an overall tool for PCIe devices which can analyze devices, debug tools for errors, validate the code.

4.2 Code to retrieve PCI device data:

```
int ret,data,vendor,device,n,interruptpin,interruptline;
```

In this program we will extend the previous program and retrieve extra more data from config space like device id, vendor id, interrupt pin, interrupt line and class code. The above statement initializes the variables required to store the data retrieved.

To retrieve this information we have to modify the address making line by adding the offset means address of the info located in config space standard header.

```
long int offset;
```

The above variable is used to store the address of info based on config space.

```
addr = PCI_BASE_ADDR | (bus << 16) | (dev << 11) | (fun << 8) | (offset & 0x00);
```

use while loop to give multiple offsets and get info based on offset provided.

The above statement gives the address based on offset and we retrieve the info from config space using inl and outl functions like before.

By using case structure and while loops we can retrieve info based on user requirement and print accordingly this saves the time and program run time.

The above code gives us the data of a PCI device such as device id, vendor id, class code, header type, subsystem vendor and device id, interrupt pin and interrupt line of the PCI device. As my project is to build a tool to debug PCI device so this code helps me to retrieve data which is required to locate a device for debugging purpose.

Through this code my PCI tool code was half way there to complete. I have used a switch case to get different data based on our choice and the output will be of all devices.

The data I got from retrieving the memory space which is configuration space of PCI devices and those space consists of the all the details of the device and io privilege gives the permission to access the kernel and configuration space.

4.3 Code to read config space (Similar to lspci source code)

This code gives the similar output but we will give the options directly through the command line arguments and based on the options provided we will get the output info similar to lspci arguments.

```
#include <stdio.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#include "pci.h"
#include <linux/pci_regs.h>
```

Include all the required libraries such as the offset stored libraries pci_regs.h which gives the predefined macros for the offset of the different information in config space and we use them to retrieve data. Simply we are modifying the offset address with name defined in macros library.

And using the command line arguments and the address making process also made simple in this program. We no need of the loop with 256 times for bus, 32 times for device and 8 times for function and get the data for all present. We will replace that mighty loop with a single statement.

```
struct pci_access *pacc;
```

The above statement gives the structure and stores all the info of all the pcie devices present in system.

```
struct pci_dev *dev;
```

The above structure gives the devices linked list and we use it for looping and get next device info from pacc structure.

```
unsigned int c,d,cap,cap1,val,sv_id,sd_id;
```

```
char classbuf[128], vendbuf[128], devbuf[128], svbuf[128], sdbuf[128];
```

```
char namebuf[1024], *name;
```

```
char i,*driver;
```

```
int opt_n=0,opt_m=0,opt_b=0,opt_c=0,count=0,opt_h=0;
```

The above statements gives the variables required to store the values got from command line and info that is retried from config space.

```
pacc = pci_alloc();
```

The above statement allocates some memory space to the structure to store the device data.

```
pci_init(pacc);
```

It initialize the pci pacc structure to store the info with all the child elements and offset elements.

```
pci_scan_bus(pacc);
```

The above statement gives the elements present in the structures a value got from scanning the device.

```
for(int j=1;j<argc;j++){
```

```
    i=*argv[j];
```

The above loop get the command line argument and separate the string and count the number of arguments given and what is the user asked by command line arguments.

```
switch (i){
    case 'n':
        opt_n++;
        break;
```

Using the above switch case, we will get the number of arguments count and detailed info of what user wants.

```
for (dev=pacc->devices; dev; dev=dev->next){
```

Using the above for loop we get the devices list based on the scanning process done at first and stored in pacc structure. now we use the same pacc structure to loop and dev structure which is a child or loop iterator. As I said before the devices are stored in linked list we iterate that list until null to get all devices list.

```
c=pci_read_byte(dev,PCI_CLASS_REVISION);
```

The above statement reads the byte value of the offset provided by the PCI_CLASS_REVISION macro with dev structure which specifies the specific device.

```
if(opt_n==1){
    if(c!=0){
        printf("%02x:%02x.%d %04x: %04x:%04x (rev
%02x)\n",dev->bus,dev->dev,dev->func,dev->device_class,dev->vendor_id,dev-
>device_id,c);
    }
    else{
        printf("%02x:%02x.%d %04x: %04x:%04x\n",dev-
>bus,dev->dev,dev->func,dev->device_class,dev->vendor_id,dev->device_id);
    }
}
```

```
}
```

The above code gives the output based on the data got from pci read api and the code prints the following only if the command line argument consists of that condition and data is not equal to FF or 00. If not it continues the loop for other condition in command line argument.

```
else if(opt_n>1){
    if(c!=0){
        printf("%02x:%02x.%d %s [%04x]: %s [%04x:%04x] (rev
%02x)\n",dev->bus,dev->dev,dev->func,pci_lookup_name(pacc, classbuf,
sizeof(classbuf),PCI_LOOKUP_CLASS,dev->device_class),dev-
>device_class,pci_lookup_name(pacc, devbuf,
sizeof(devbuf),PCI_LOOKUP_VENDOR | PCI_LOOKUP_DEVICE,dev->vendor_id,
dev->device_id),dev->vendor_id,dev->device_id,c);
    }
}
```

The above code prints the following if the command line arguments have more than one options in the string which gives more info on the devices like more specific. And uses the same logic as the previous one.

```
else{
    printf("%02x:%02x.%d %s [%04x]: %s
[%04x:%04x]\n",dev->bus,dev->dev,dev->func,pci_lookup_name(pacc, classbuf,
sizeof(classbuf),PCI_LOOKUP_CLASS,dev->device_class),dev-
>device_class,pci_lookup_name(pacc, devbuf,
sizeof(devbuf),PCI_LOOKUP_VENDOR | PCI_LOOKUP_DEVICE,dev->vendor_id,
dev->device_id),dev->vendor_id,dev->device_id);
}
}
```

If the command line argument is not meet with the f condition then the else condition which goes for the next option comes up to check the condition.

```

else if(opt_m>=1){

    printf("%02x:%02x.%d\t",dev->bus,dev->dev,dev->func);

    sv_id=pci_read_byte(dev,PCI_SUBSYSTEM_VENDOR_ID);

    sd_id=pci_read_byte(dev,PCI_SUBSYSTEM_ID);

    printf(" \"%s\" ",pci_lookup_name(pacc, classbuf,
sizeof(classbuf), PCI_LOOKUP_CLASS, dev->device_class));

    printf(" \"%s\" ",pci_lookup_name(pacc, vendbuf, sizeof(vendbuf),
PCI_LOOKUP_VENDOR, dev->vendor_id, dev->device_id));

    printf(" \"%s\" ",pci_lookup_name(pacc, devbuf, sizeof(devbuf),
PCI_LOOKUP_DEVICE, dev->vendor_id, dev->device_id));

    printf(" \"%s\" ",pci_lookup_name(pacc, sdbuf, sizeof(sdbuf),
PCI_LOOKUP_SUBSYSTEM | PCI_LOOKUP_DEVICE, dev->vendor_id, dev-
>device_id, sv_id, sd_id));

    c=pci_read_byte(dev,PCI_CLASS_REVISION);

    printf("\r-r%x\" ",c);

    c=pci_read_byte(dev,PCI_CLASS_PROG);

    printf("\r-p%x\" \n",c);

}

```

The above code gives the output of the option m in command line argument and all the details what this option represents will be in man page of this driver when installed in kernel.

After all the conditions meet with the condition then default condition is used which shows invalid option.

```

else if(opt_b>=1){
    if(c!=0){
        printf("%02x:%02x.%d %s [%04x]: %s (rev %02x)\n",dev-
>bus,dev->dev,dev->func,pci_lookup_name(pacc,      classbuf,
sizeof(classbuf),PCI_LOOKUP_CLASS,dev->device_class),dev-
>device_class,pci_lookup_name(pacc,                      devbuf,
sizeof(devbuf),PCI_LOOKUP_VENDOR | PCI_LOOKUP_DEVICE,dev->vendor_id,
dev->device_id),c);
    }
    else{
        printf("%02x:%02x.%d %s [%04x]: %s\n",dev->bus,dev-
>dev,dev->func,pci_lookup_name(pacc,      classbuf,
sizeof(classbuf),PCI_LOOKUP_CLASS,dev->device_class),dev-
>device_class,pci_lookup_name(pacc,                      devbuf,
sizeof(devbuf),PCI_LOOKUP_VENDOR | PCI_LOOKUP_DEVICE,dev->vendor_id,
dev->device_id));
    }
}

```

The above option shows the output of the option b and this gives the most sensitive information and in fact all the details which is more than enough to debug a tool.

Based on this info and status register present in the config space we will decide what is the problem occurred in pci tool and debug and make log file.

After the log file the troubleshoot tool makes the device to solve its error or gives the user about the error so that user can understand what is the problem prior it becomes toxic.

```
else if(opt_c==1){
```

```
    count=0;
```

```
    d=pci_read_byte(dev,PCI_CAPABILITY_LIST);
```

This option c is a special one which specifies about the capability register and its capabilities of the devices as the device may contain multiple capabilities like power management, message interrupts, message signals, accelerated graphics port etc.

Since the capabilities details are stored in linked list as explained in previous chapters the retrieving of the capability register is tough. It is done by retrieving each by each capability and goes on again to retrieve the characteristics of that capability.

```
    cap1=d;
```

```
    if(d!=0){
```

```
        printf("%04x:%02x:%02x.%d\n",dev->domain, dev->bus,  
dev->dev, dev->func);
```

```
        while(d!=0){
```

```
            printf("Capability Address %d = %x\n",count+1,d);
```

```
            val=pci_read_byte(dev,d);
```

Reads the capability pointer address linked list and gets a value of the type and based on the type of capability the characteristics are read.

```
            printf("Capability ID %d = %x ",count+1,val);
```

```
            if(val==PCI_CAP_ID_PM)
```

```
                printf("(Power Management)\n");
```

```
            else if(val==PCI_CAP_ID_PCIX)
```

```
                printf("(PCI-X non-Bridge)\n");
```

```
            else if(val==PCI_CAP_ID_SATA)
```

```
                printf("(SATA)\n");
```

```
            d=pci_read_byte(dev,d+1);
```

```
count++;
```

```
}
```

After the characteristics are read then the value is read by getting a special address as shown in below to get the value for the other characteristic.

```
cap = pci_read_word(dev, cap1 + PCI_CAP_FLAGS);
```

```
printf("Power Management Version %d\n", cap &  
PCI_PM_CAP_VER_MASK);
```

```
printf("_____ \n");
```

```
}
```

```
}
```

```
else{
```

```
break;
```

```
}
```

```
}
```

The below if condition gives when the user is needed for a man page since it is a c code we made it in a option of command line argument but the next step of this program is to make a kernel driver tool and then in that we can make this If statement in man page directly without this option.

```
if(opt_h==1){
```

```
printf("Options \t Function\n");
```

```
printf(" m    Dump PCI device data in a backward-compatible  
machine readable form. See below for details.\n mm    Dump PCI device data in a  
machine readable form for easy parsing by scripts.\n b    Bus-centric view. Show all  
IRQ numbers and addresses as seen by the cards on the PCI bus instead of as seen  
by the kernel.\n D    Always show PCI domain numbers. By default, lspci suppresses  
them on machines which have only domain 0.\n P    Identify PCI devices by path
```

through each bridge, instead of by bus number.\n PP Identify PCI devices by path through each bridge, showing the bus number as well as the device number.\n n Show PCI vendor and device codes as numbers instead of looking them up in the PCI ID list.\n nn Show PCI vendor and device codes as both numbers and names.\n c Display the devices with capabilities with its address, ID and other info.\n h Man Page.\n");

}

pci_cleanup(pacc);

return 0;

}

CHAPTER-5: RESULT AND ANALYSIS

```
Activities Terminal Mar 11 17:06
tejanaveen@teja-HCL: ~
tejanaveen@teja-HCL:~$ vi lspci.c
tejanaveen@teja-HCL:~$ gcc lspci.c -o lspci
tejanaveen@teja-HCL:~$ ./lspci
bus # dev # fun #
lopl set error: Operation not permitted
tejanaveen@teja-HCL:~$ sudo ./lspci
[sudo] password for tejanaveen:
bus # dev # fun #
0 0 0
0 1 0
0 1 1
0 2 0
0 3 0
0 4 0
0 5 0
0 6 0
0 7 0
0 d 0
tejanaveen@teja-HCL:~$ lspci
00:00:0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Native] (rev 02)
00:01:0 ISA bridge: Intel Corporation 82375B PIIX3 ISA [Native/Triton II]
00:01:1 IDE interface: Intel Corporation 8237AB/EB/MB PIIX4 IDE (rev 01)
00:02:0 VGA compatible controller: VMware SVGA II Adapter
00:03:0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:04:0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest Service
00:05:0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:06:0 USB controller: Apple Inc. KeyLargo/Interpreid USB
00:07:0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d:0 SATA controller: Intel Corporation 82801HM/HM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
tejanaveen@teja-HCL:~$ grep PCI_ID /sys/bus/pci/devices/*/uevent
/sys/bus/pci/devices/0000:00:01:0/uevent:PCI_ID=0080:7000
/sys/bus/pci/devices/0000:00:01:1/uevent:PCI_ID=0080:7111
/sys/bus/pci/devices/0000:00:02:0/uevent:PCI_ID=0080:0405
/sys/bus/pci/devices/0000:00:03:0/uevent:PCI_ID=0080:100E
/sys/bus/pci/devices/0000:00:04:0/uevent:PCI_ID=0080:0CAF
/sys/bus/pci/devices/0000:00:05:0/uevent:PCI_ID=0080:2415
/sys/bus/pci/devices/0000:00:06:0/uevent:PCI_ID=0080:003F
/sys/bus/pci/devices/0000:00:07:0/uevent:PCI_ID=0080:7113
/sys/bus/pci/devices/0000:00:0d:0/uevent:PCI_ID=0080:2829
tejanaveen@teja-HCL:~$
```

Figure 5.1 – lspci code execution

```
Mar 4 15:55
tejanaveen@teja-HCL:~
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00: 86 80 13 71 01 80 80 02 08 00 80 06 00 00 80 00
10: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
00: 86 80 28 87 00 10 00 02 01 80 01 00 00 00 00 00
10: 41 d2 00 00 49 d2 00 00 51 d2 00 00 59 d2 00 00
20: 61 d2 00 00 60 80 f8 00 00 00 00 00 00 00 00 00
30: 00 00 00 00 70 00 00 00 00 00 00 00 00 01 00 00
tejanaveen@teja-HCL:~$ lspci -nn
00:00.0 Host bridge [8080]: Intel Corporation 440FX - 82441FX PMC [Natoma] [8086:1237] (rev 02)
00:01.0 ISA bridge [8001]: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II] [8086:7000]
00:01.1 IDE interface [8010]: Intel Corporation 82371AB/EB/MB PIIX4 IDE [8086:7111] (rev 01)
00:02.0 VGA compatible controller [0300]: VMware SVGA II Adapter [15ad:0405]
00:03.0 Ethernet controller [0200]: Intel Corporation 82540EM Gigabit Ethernet Controller [8086:100e] (rev 02)
00:04.0 System peripheral [0800]: Innoket Systemberatung GMBH VirtualBox Guest Service [80ee:cafe]
00:05.0 Multimedia audio controller [0401]: Intel Corporation 82801AA AC'97 Audio Controller [8086:2415] (rev 01)
00:06.0 USB controller [0c03]: Apple Inc. Keyboard/Intrepid USB [106b:003f]
00:07.0 Bridge [8080]: Intel Corporation 82371AB/EB/MB PIIX4 ACPI [8086:7111] (rev 08)
00:0d.0 SATA controller [0160]: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] [8086:2829] (rev 02)
tejanaveen@teja-HCL:~$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: VMware SVGA II Adapter
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:04.0 System peripheral: Innoket Systemberatung GMBH VirtualBox Guest Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:06.0 USB controller: Apple Inc. Keyboard/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
tejanaveen@teja-HCL:~$ lspci -n
00:00.0 8080: 8086:1237 (rev 02)
00:01.0 8001: 8086:7000
00:01.1 8010: 8086:7111 (rev 01)
00:02.0 0300: 15ad:0405
00:03.0 0200: 8086:100e (rev 02)
00:04.0 0800: 80ee:cafe
00:05.0 0401: 8086:2415 (rev 01)
00:06.0 0c03: 106b:003f
00:07.0 8080: 8086:7111 (rev 08)
00:0d.0 0160: 8086:2829 (rev 02)
tejanaveen@teja-HCL:~$
```

Figure 5.2 – Standard lspci command execution

```

tejanaveen@teja-HCL:~$ gcc io.c -o io
tejanaveen@teja-HCL:~$ sudo ./io
[sudo] password for tejanaveen:
Enter your choice:
1)Retrieve Device and Vendor ID
2)Retrieve Class Code
3)Retrieve Subsystem vendor and device ID
4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 1
bus # dev # fun # vendor id # device id #
0 0 0 8086 1237
0 1 0 8086 7000
0 1 1 8086 7111
0 2 0 15ad 405
0 3 0 8086 100e
0 4 0 8086 cafe
0 5 0 8086 2415
0 6 0 1060 3f
0 7 0 8086 7113
0 d 0 8086 2829
Enter your choice:
1)Retrieve Device and Vendor ID
2)Retrieve Class Code
3)Retrieve Subsystem vendor and device ID
4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 2
bus # dev # fun # class-code #
0 0 0 609
0 1 0 601
0 1 1 101
0 2 0 309
0 3 0 209
0 4 0 880
0 5 0 401
0 6 0 c03
0 7 0 680
0 d 0 106
Enter your choice:
1)Retrieve Device and Vendor ID
2)Retrieve Class Code
3)Retrieve Subsystem vendor and device ID
4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 3
bus # dev # fun # vendor # device #

```

Figure 5.3 – Code execution of IO mapped code

```

4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 3
bus # dev # fun # vendor # device #
0 2 0 15ad 405
0 3 0 8086 1e
0 5 0 1028 177
Enter your choice:
1)Retrieve Device and Vendor ID
2)Retrieve Class code
3)Retrieve Subsystem vendor and device ID
4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 4
bus # dev # fun # Header type #
0 1 0 80
0 1 1 0
0 2 0 0
0 3 0 0
0 5 0 0
0 6 0 0
0 7 0 80
0 d 0 0
Enter your choice:
1)Retrieve Device and Vendor ID
2)Retrieve Class Code
3)Retrieve Subsystem vendor and device ID
4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 5
bus # dev # fun # Interrupt Pin # Interrupt Line #
0 2 0 1 a
0 3 0 1 9
0 4 0 1 b
0 5 0 1 b
0 6 0 1 a
0 7 0 1 9
0 d 0 1 b
Enter your choice:
1)Retrieve Device and Vendor ID
2)Retrieve Class Code
3)Retrieve Subsystem vendor and device ID
4)Retrieve Header Type
5)Retrieve IRQ
6)Exit: 6
tejanaveen@teja-HCL:~$

```

Figure 5.4 – Code execution of IO mapped code

```

Activities Terminal Apr 1 11:59
tejanaveen@teja-HCL: ~
tejanaveen@teja-HCL:~$ ./scampcl D
00:05.0 Multimedia audio controller [0401]: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:02.0 VGA compatible controller [0300]: VMware SVGA II Adapter
00:06.0 USB controller [0c03]: Apple Inc. KeyLargo/Intrepid USB
00:01.1 IDE interface [0101]: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:03.0 Ethernet controller [0200]: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:00.0 Host bridge [0600]: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:07.0 Bridge [0680]: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:04.0 System peripheral [0880]: Innatek Systemberatung GmbH VirtualBox Guest Service
00:01.0 ISA bridge [0601]: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:0d.0 SATA controller [0106]: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
tejanaveen@teja-HCL:~$ ./scampcl k
[sudo] password for tejanaveen:
0000:00:03.0
Capability Address 1 = dc
Capability ID 1 = 1 (Power Management)
Capability Address 2 = e4
Capability ID 2 = 7 (PCI-X non-Bridge)
Power Management Version 2
.....
0000:00:0d.0
Capability Address 1 = 70
Capability ID 1 = 1 (Power Management)
Capability Address 2 = a8
Capability ID 2 = 12 (SATA)
Power Management Version 3
.....
tejanaveen@teja-HCL:~$ ./scampcl h
Options Functions
n Dump PCI device data in a backward-compatible machine readable form. See below for details.
nm Dump PCI device data in a machine readable form for easy parsing by scripts.
b Bus-centric view. Show all IRQ numbers and addresses as seen by the cards on the PCI bus instead of as seen by the kernel.
D Always show PCI domain numbers. By default, lspci suppresses them on machines which have only domain 0.
P Identify PCI devices by path through each bridge, instead of by bus number.
PP Identify PCI devices by path through each bridge, showing the bus number as well as the device number.
n Show PCI vendor and device codes as numbers instead of looking them up in the PCI ID list.
nn Show PCI vendor and device codes as both numbers and names.
c Display the devices with capabilities with its address, ID and other info.
h Run Help.
tejanaveen@teja-HCL:~$

```

```

Activities Terminal Apr 1 11:59
tejanaveen@teja-HCL: ~
tejanaveen@teja-HCL:~$ gcc code.c -Iproject/pciutils/lib/ -lpci -o scampcl
tejanaveen@teja-HCL:~$ ./scampcl n
00:05.0 0401: 8086:2415 (rev 01)
00:02.0 0300: 15ad:0405
00:06.0 0c03: 100b:003f
00:01.1 0101: 8086:7111 (rev 01)
00:03.0 0200: 8086:100e (rev 02)
00:00.0 0600: 8086:1237 (rev 02)
00:07.0 0680: 8086:7113 (rev 08)
00:04.0 0880: 80ee:cafe
00:01.0 0601: 8086:7800
00:0d.0 0106: 8086:2829 (rev 02)
tejanaveen@teja-HCL:~$ ./scampcl n n
00:05.0 Multimedia audio controller [0401]: Intel Corporation 82801AA AC'97 Audio Controller [8086:2415] (rev 01)
00:02.0 VGA compatible controller [0300]: VMware SVGA II Adapter [15ad:0405]
00:06.0 USB controller [0c03]: Apple Inc. KeyLargo/Intrepid USB [100b:003f]
00:01.1 IDE interface [0101]: Intel Corporation 82371AB/EB/MB PIIX4 IDE [8086:7111] (rev 01)
00:03.0 Ethernet controller [0200]: Intel Corporation 82540EM Gigabit Ethernet Controller [8086:100e] (rev 02)
00:00.0 Host bridge [0600]: Intel Corporation 440FX - 82441FX PMC [Natoma] [8086:1237] (rev 02)
00:07.0 Bridge [0680]: Intel Corporation 82371AB/EB/MB PIIX4 ACPI [8086:7113] (rev 08)
00:04.0 System peripheral [0880]: Innatek Systemberatung GmbH VirtualBox Guest Service [80ee:cafe]
00:01.0 ISA bridge [0601]: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II] [8086:7800]
00:0d.0 SATA controller [0106]: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] [8086:2829] (rev 02)
tejanaveen@teja-HCL:~$ ./scampcl n
00:05.0 "Multimedia audio controller" "Intel Corporation" "82801AA AC'97 Audio Controller" "-r1" "-p0"
00:02.0 "VGA compatible controller" "VMware" "SVGA II Adapter" "SVGA II Adapter" "-r0" "-p0"
00:06.0 "USB controller" "Apple Inc." "KeyLargo/Intrepid USB" "KeyLargo/Intrepid USB" "-r0" "-p0"
00:01.1 "IDE interface" "Intel Corporation" "82371AB/EB/MB PIIX4 IDE" "82371AB/EB/MB PIIX4 IDE" "-r1" "-paa"
00:03.0 "Ethernet controller" "Intel Corporation" "82540EM Gigabit Ethernet Controller" "82540EM Gigabit Ethernet Controller" "-r2" "-p0"
00:00.0 "Host bridge" "Intel Corporation" "440FX - 82441FX PMC [Natoma]" "440FX - 82441FX PMC [Natoma]" "-r2" "-p0"
00:07.0 "Bridge" "Intel Corporation" "82371AB/EB/MB PIIX4 ACPI" "82371AB/EB/MB PIIX4 ACPI" "-r8" "-p0"
00:04.0 "System peripheral" "Innatek Systemberatung GmbH" "VirtualBox Guest Service" "VirtualBox Guest Service" "-r0" "-p0"
00:01.0 "ISA bridge" "Intel Corporation" "82371SB PIIX3 ISA [Natoma/Triton II]" "82371SB PIIX3 ISA [Natoma/Triton II]" "-r0" "-p0"
00:0d.0 "SATA controller" "Intel Corporation" "82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]" "82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode]" "-r2" "-p1"
tejanaveen@teja-HCL:~$ ./scampcl b
00:05.0 Multimedia audio controller [0401]: Intel Corporation 82801AA AC'97 Audio Controller (rev 01)
00:02.0 VGA compatible controller [0300]: VMware SVGA II Adapter
00:06.0 USB controller [0c03]: Apple Inc. KeyLargo/Intrepid USB
00:01.1 IDE interface [0101]: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:03.0 Ethernet controller [0200]: Intel Corporation 82540EM Gigabit Ethernet Controller (rev 02)
00:00.0 Host bridge [0600]: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:07.0 Bridge [0680]: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:04.0 System peripheral [0880]: Innatek Systemberatung GmbH VirtualBox Guest Service
00:01.0 ISA bridge [0601]: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:0d.0 SATA controller [0106]: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA Controller [AHCI mode] (rev 02)
tejanaveen@teja-HCL:~$

```

Fig 5.5 Code Execution to read config space

REFERENCES

- 1) <https://nixhacker.com/playing-with-pci-device-memory/>
- 2) http://www.newfreesoft.com/programming/c_language_traverse_pci_device_1598/
- 3) <https://www.kernel.org/doc/html/latest/driver-api/pci/pci.html>
- 4) https://www.tutorialspoint.com/unix_commands/lspci.htm
- 5) **Book:** PCI Express System Architecture by Ravi Budruk, Don Anderson, Tom Shanley – MindShare Inc. – 2008 – 41 to 67
- 6) PCI Express Basics and Background by Richard Solomon – 2014 published by PCI-SIG.
- 7) <https://github.com/brgl/busybox/blob/master/util-linux/lspci.c>
- 8) <https://fossies.org/linux/pciutils/example.c>
- 9) https://sites.uclouvain.be/SystInfo/usr/include/linux/pci_regs.h.html
- 10) <https://github.com/gittup/pciutils/blob/gittup/lspci.c>
- 11) [PCI Express BAR memory mapping basic understanding \(newbedev.com\)](https://newbedev.com/pci-express-bar-memory-mapping-basic-understanding/)
- 12) [PCI BARs and other means of accessing the GPU — envytools gitdocumentation](https://envytools.gitdocumentation.com/pci-bars-and-other-means-of-accessing-the-gpu/)
- 13) [PCI vs PCI-X vs PCI-E, Why Choose PCI-E Card? | FS Community](https://fscommunity.com/pci-vs-pci-x-vs-pci-e-why-choose-pci-e-card/)
- 14) <https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-controller-i350-datasheet.pdf>
- 15) [ls-caps.c - pub/scm/utls/pciutils/pciutils - Git at Google \(googlesource.com\)](https://pub/scm/utls/pciutils/pciutils)
- 16) [PCI Configuration Space - Intel Communities](https://www.intel.com/comm/pci/configuration-space/)
- 17) [Microsoft Word - PCI Code-ID r 1 11-draft- v24 Jan 2019.docx\(pcisig.com\)](https://www.pcisig.com/wp-content/uploads/2019/01/PCI_Code-ID_r1_11-draft-v24_Jan_2019.docx)