# 2. PCI EXPRESS BUS

This laboratory work presents the serial variant of the PCI bus, referred to as PCI Express. After an overview of the PCI Express bus, details about its architecture are presented, including the PCI Express link, bus topology, architectural layers, transactions, and interrupts. The physical layer is presented in more detail and the most important configuration registers are described. The aim of the applications is to access the PCI configuration space and to decode the information available in the configuration registers of PCI and PCI Express devices.

## 2.1. Overview of PCI Express Bus

The PCI Express (PCIe) bus represents the third generation of the PCI (*Peripheral Component Interconnect*) bus, with higher performance and reliability compared to the previous generations PCI and PCI-X. As opposed to these previous versions, which are parallel buses, PCIe is a serial bus. Due to the serial nature of the PCIe bus, it has several advantages compared to a parallel bus: lower pin count of the integrated circuits, lower complexity, and lower cost of the printed circuit boards.

The PCIe bus specifications originate from the specifications of the 3GIO (*Third Generation I/O*) bus, which have been developed by the *Arapahoe Work Group*. This group was composed of representatives from the companies Compaq Computer, Dell Computer, Hewlett-Packard, IBM, Intel, and Microsoft. The specifications have been transferred in 2002 to the PCI-SIG (*PCI Special Interest Group*), a group of over 900 companies that has developed and updated the standards of various versions of the previous buses PCI and PCI-X (www.pcisig.com). The new bus has been renamed PCI Express, a name which reflects the high speed of the bus, as well as its software compatibility with the previous generations PCI and PCI-X. Figure 2.1 illustrates the logo of the PCI Express bus.

**Figure 2.1.** Logo of PCI Express bus.

The designers of the PCIe bus have maintained the main advantageous features of the architecture of previous PCI bus generations. For instance, the PCIe bus uses the same communication model as the PCI and PCI-X buses. The same address spaces are retained: memory, I/O, and configuration. The PCIe bus allows using the same types of transactions as the previous buses: memory read/write, I/O read/write, and configuration read/write. This way, compatibility is maintained with existing operating systems and software drivers, which do not require changes.

Like previous PCI buses, PCIe supports chip-to-chip interconnection and board-to-board interconnection via expansion cards and connectors. The expansion cards have a structure similar to that used by the expansion cards of PCI and PCI-X buses. A PCIe motherboard has a similar form factor to existing ATX motherboards, used for personal computers.

In addition to retaining some advantageous features of the PCI and PCI-X buses, the PCIe bus introduces various improvements for enhancing performance and reducing cost. As opposed to the previous PCI and PCI-X generations, which are shared parallel buses, the PCIe bus uses a serial point-to-point interconnect for communication between two peripheral devices. First, a serial interconnect eliminates the disadvantages of a parallel bus, especially the difficulty of synchronization between multiple data lines due to the asymmetrical signal propagation (skew). The cause of this skew may be the different length of data paths traveled by various signals or the propagation on different layers of the printed circuit board. Although the data signals of a parallel bus are transmitted simultaneously, they may reach the destination at different times. Increasing the clock frequency of a parallel bus is difficult, since the clock cycle time may become shorter than the signal skew (which can be of a few nanoseconds). For a serial bus the signal skew problem does not arise, because there is no external clock signal, as the synchronization information is embedded into the transmitted serial signal. Second, a point-to-point interconnect implies a reduced electrical load of the link, which enables to increase the frequency of the clock signal used for data transfers.

The performance of PCIe bus is scalable, which is obtained by implementing a variable number of communication lanes per interconnect, based on performance requirements for that interconnect.

The PCIe bus implements switch-based technology to interconnect a large number of peripheral devices. For the serial interconnect a packet-based communication protocol is used. Instead of special signals for various functions, such as interrupt signaling, error handling, or power management, both data and commands are transmitted in packets. By this the pin count of devices and their cost are reduced.

The PCIe bus has several advanced features. For instance, the *Quality of Service* (QoS) feature allows to ensure differentiated performance for different applications. The hot plug and hot swap support enables to build systems that are always available. Advanced power management features allow to implement mobile applications with low power consumption. The error handling feature makes the PCIe bus suitable for robust systems required for high-end servers.

## 2.2. PCI Express Bus Features

The main features of the PCIe bus are the following:

- It unifies the I/O architecture for different types of systems, such as desktop computers, mobile computers, workstations, servers, communication platforms, and embedded systems.
- It enables to interconnect integrated circuits on the motherboard, as well as expansion cards via connectors or cables.
- The communication is based on packets, with a high transfer rate and efficiency.
- The interface is serial, which enables to reduce the pin count and to simplify the interconnections.
- Performance is scalable, which is obtained through the ability to implement a particular interconnect via several communication lanes.
- The software model is compatible with the classical PCI architecture, which allows to configure PCIe devices, loading operating systems, and using existing software drivers, without the need for changes.
- It provides a differentiated quality of service (QoS) through the ability to allocate dedicated resources for certain data flows, to configure the QoS arbitration policies for each component, and to use isochronous transfers for real-time applications.
- It provides an advanced power management through the ability to identify power management capabilities of each peripheral device, to transition a peripheral device into a state with a specific power consumption, and to receive notifications of the current power state of a peripheral device.
- It ensures link-level data integrity for all types of transactions.

- It supports advanced error reporting and handling to improve fault isolation and error recovery.
- It supports hot-plug and hot-swap of peripheral devices, without the need to use additional signals.

## 2.3. PCI Express Bus Architecture

### 2.3.1. PCI Express Bus Link

A minimal PCIe link consists of two unidirectional (simplex) communication channels between two PCIe peripheral devices, one channel for transmit and one for receive (Figure 2.2). Data and command packets are sent over these channels. Each channel is implemented physically through a pair of wires over which low-voltage differential signals are transmitted. Such a minimal PCIe link is called communication *lane*. To scale bandwidth, a PCIe link may aggregate multiple communication lanes, denoted by xN, where N is the link width. The PCIe bus specification indicates the possibility of using link widths of x1, x2, x4, x8, x12, x16, and x32.

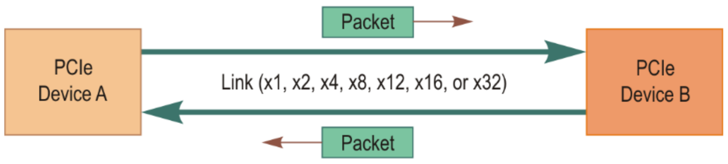

**Figure 2.2.** PCI Express link.

During hardware initialization, for each PCIe link the lane width and frequency of operation are negotiated. The link width and frequency of operation are set automatically by the devices at each end of the link, without involving the operating system. After initialization, each link must only operate at the operating frequency that has been set. The first version of the PCIe specification defined an operating frequency of 2.5 GHz, which corresponds to an effective bandwidth of 2.5 Gbits/s for each communication lane and direction. In the subsequent versions, the operating frequency increased to 5 GHz, and then to 8 GHz.

### 2.3.2. PCI Express Bus Topology

A PCIe system is comprised of PCIe links that interconnect a set of components. An example topology is illustrated in Figure 2.3. The main components of this topology are a root complex, multiple endpoints (I/O devices), a switch, and a PCIe-PCI bridge, all interconnected via PCIe links. All devices and links associated with a root complex, which are connected to it directly or indirectly (via switches and bridges) represent a *hierarchy*.

The *root complex* is the device that connects one or more processors and the memory subsystem to the I/O devices. This device represents the root of an I/O hierarchy. The root complex may support one or more PCIe ports; in Figure 2.3, the root complex contains three ports. Each port defines a separate *hierarchy domain*. Each hierarchy domain may be composed of a single endpoint or a sub-hierarchy containing one or more switches and endpoints.

A root complex implements various resources, such as interrupt controller, power management controller, error detection and reporting logic. The root complex contains an internal bus, which represents the bus number 0 in the entire hierarchy. This device initiates transaction requests on behalf of a processor, transmits packets out of its ports and receives packets on its ports which it transmits to memory. Optionally, a multi-port root complex may also route packets from one port to another port.

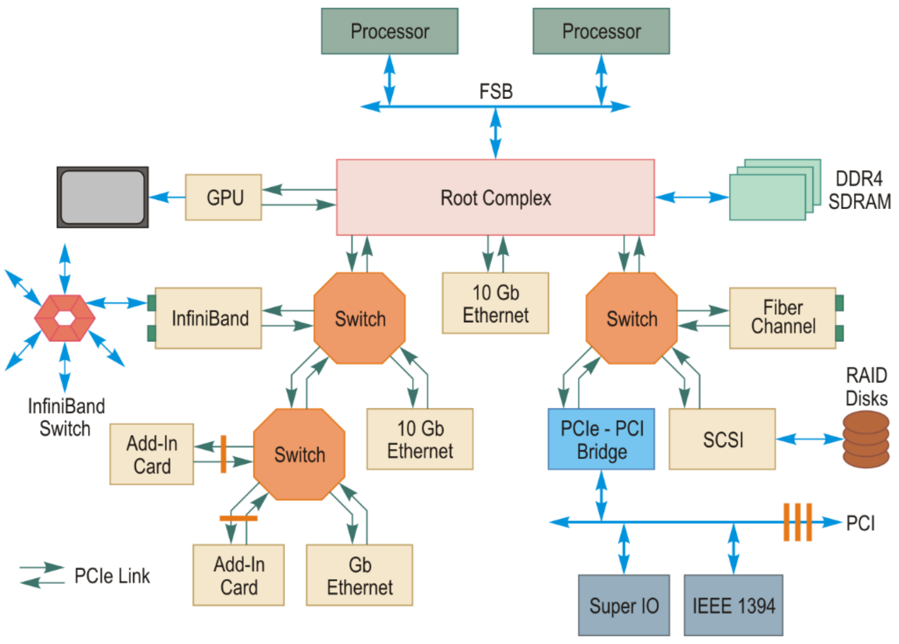

**Figure 2.3.** Example PCI Express topology.

*Endpoints* represent peripheral devices that participate to PCIe transactions. There are two types of endpoints. An *initiator* (requester) endpoint initiates a transaction in the PCIe system, while a *target* (completer) endpoint responds to transactions that are addressed to it. In a PCIe hierarchy, in addition to PCIe endpoints, legacy endpoints may also exist, which are compatible with previous generations of the PCI bus. Like with the classical PCI bus, PCIe devices may have up to eight logical functions, so that an endpoint may be composed of up to eight functions numbered from 0 through 7. Each endpoint is assigned a device identifier (ID), which consists of a bus number, device number, and function number.

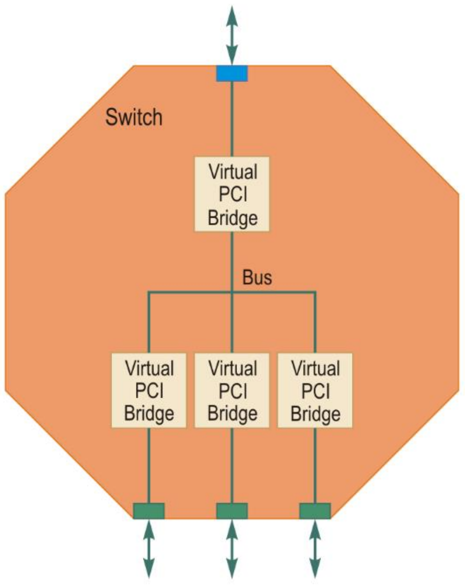

**Figure 2.4.** Internal structure of a switch.

A *switch* is defined as a logical assembly of multiple virtual PCI-to-PCI bridges, each bridge associated with a switch port. The switch in Figure 2.4 consists of four virtual bridges. These bridges are connected via an internal bus. One port of the switch is connected to the root complex, and the other ports are connected to endpoints or other switches. To configuration software, a switch appears as two or more logical PCI-to-PCI bridges.

A switch forwards packets from any of its input (ingress) ports to one of its output (egress) ports, in a manner similar to a PCI-to-PCI bridge. The packets are transferred via a routing mechanism based on either an address or an identifier. An arbitration mechanism is used, by which the priority with which packets are forwarded from input ports to output ports is determined.

### 2.3.3. PCI Express Architecture Layers

A PCIe system may be structured into five logical layers, which are described in short next.

- The *configuration/OS layer* manages the configuration of PCIe devices by the operating system based on the *Plug-and-Play* specifications for initializing, enumerating, and configuring I/O devices.
- The *software layer* interacts with the operating system through the same drivers as the conventional PCI bus.
- The *transaction layer* manages the transmission and reception of information using a packet-based protocol.
- The *data link layer* ensures the integrity of data transfers via error detection using a *Cyclic Redundancy Check* (CRC).
- The *physical layer* performs packet transmission over the PCIe serial links.

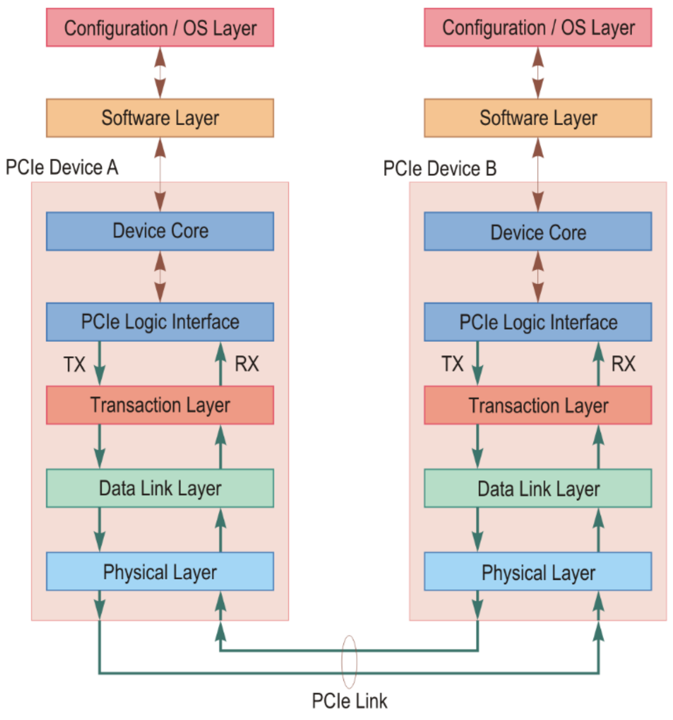

**Figure 2.5.** Layers of a PCIe system and PCIe device.

The PCIe specification defines the architecture of PCIe devices in terms of three logical layers, which are the last three layers from those previously listed. Each of these layers may be divided into two sections, one that processes information to be transmitted and one that processes information received (Figure 2.5). This logical organization, however, does not imply a particular implementation of PCIe devices.

The PCIe bus uses packets for transferring information between pairs of devices connected via a PCIe link. Consider first the transfer of information from device A to device B. Packets are formed in the transaction layer based on information obtained from the device core and application. A particular packet is stored in a buffer to be transmitted to the lower layers. The data link layer extends the packet with additional information required for error detection at a receiver device. This packet is then encoded in the physical layer and transmitted through differential signals over the PCIe link by the analog portion of this layer.

Now consider the reception of information by device B. Packets are decoded in the physical layer and their contents are forwarded to the upper layers. The data link layer checks for errors in a received packet, and if there are no errors forwards the packet to the transaction layer. This layer stores the packet in a buffer and converts the information in the packet to a representation that can be processed by the device core and application.

Figure 2.6 illustrates the conceptual information flow that is transferred through the three logical layers of PCIe devices.

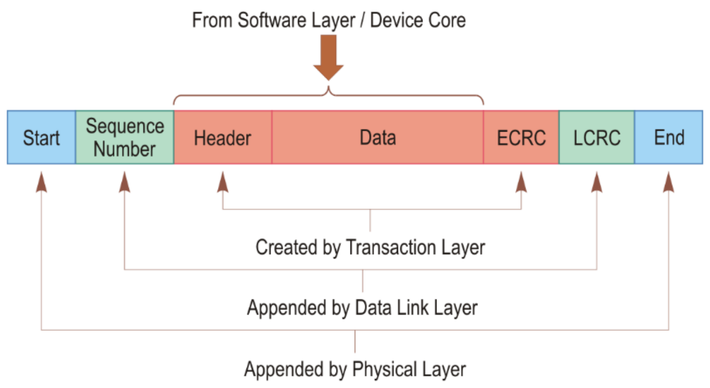
From Software Layer / Device Core
Start
Sequence Number
Header
Data
ECRC
LCRC
End
Created by Transaction Layer
Appended by Data Link Layer
Appended by Physical Layer

**Figure 2.6.** Packet flow through the logical layers of PCIe devices.

The software layer or device core sends to the transaction layer the information required to create the main section of the packet. This information is the header and data field of the packet. Optionally, a CRC code is computed and appended to the packet as the ECRC (*End-to-End* CRC) field. This field is used by the target device of the packet to detect CRC errors in the header and data field.

The packet created by the transaction layer is forwarded to the data link layer, which appends to this packet a sequence number and another CRC field, LCRC (*Link* CRC). The LCRC field is used by the receiver device at the other end of the link to detect CRC errors in the packet created by the transaction layer and in the sequence number. The resulting packet is forwarded to the physical layer, which concatenates two Start and End characters of one byte each that will frame the packet. The packet is then encoded and is transmitted through differential signals over a PCIe link using the available number of communication lanes.

### 2.3.4. PCI Express Transactions

A *transaction* is defined as a series of one or more packet transmissions required to accomplish an information transfer between an initiator and a target device. There are four categories of PCIe transactions: memory, I/O, configuration, and message. The first three categories were also supported by the previous PCI and PCI-X buses. Examples of such transactions are reading and writing the memory, reading and writing the I/O space, reading and writing the configuration registers. Message transactions are specific to the PCIe bus. Message transactions, also called messages, are used for interrupt signaling, power management, or error signaling.

On the other hand, in the PCIe architecture there are two types of transactions. The first type is represented by transactions for which the target device returns a completion packet back to the initiator device, as response to the request packet transmitted by the initiator; these are called *non-posted* transactions. These transactions are performed according to the protocol defined for split transactions supported by the PCI-X bus. With split transactions, after initiating a transaction, the target device stores the information needed for performing the transaction and signals a delayed response. The initiator device releases the bus, which will be available for other transactions. If data have been requested from the target device, such as in the case of a read transaction, the target device gathers these data, obtains bus ownership, and returns the requested data. The completion packet returned by the target device confirms that the request packet has been received by the target device.

The second type is represented by transactions for which the target device does not return a completion packet back to the initiator device; these are called *posted* transactions. In this way, the time required for completing the transaction is shorter, but the initiator device does not have knowledge of successful reception of the request packet by the target device.

### 2.3.5. PCI Express Interrupts

Devices connected to the PCIe bus may signal interrupt requests using one of two available methods. PCIe devices must use the native mechanism of the PCIe bus for interrupt signaling, that of *Message Signaled Interrupts* (MSI). Devices compatible with the previous bus generations PCI and PCI-X may use the dedicated signals of the PCI bus for interrupt requests.

The native mechanism of the PCIe bus for interrupt signaling (MSI) has been defined in version 2.2 of the PCI bus specification as an optional mechanism and became mandatory for the PCI-X bus. The term "*message signaled interrupt*" may create confusion in the context of the PCIe bus because of the existence of PCIe message transactions. A message signaled interrupt does not represent a PCIe message, instead it is simply a memory write transaction. A memory write transaction associated with the MSI mechanism can only be distinguished from other memory write transactions by its target address, the memory addresses for interrupt signaling being reserved by the system.

The mechanism compatible with the PCI bus for interrupt signaling (legacy mechanism) implies using the interrupt request signals defined for the PCI bus. These signals are *INTA#*, *INTB#*, *INTC#*, and *INTD#* (the # symbol denotes an active-low signal). Acknowledgement of an interrupt request is indicated via a certain configuration on the PCI bus control lines. A PCI-compatible device will assert one of the *INTx#* lines to signal an interrupt request. Since the PCIe bus does not include the *INTx#* interrupt lines, special messages are used that act as virtual *INTx#* lines. These messages target the interrupt controller, typically located within the root complex.

Figure 2.7 illustrates the signaling of interrupts generated by three types of devices. The PCIe device uses the native MSI mechanism. The PCI-X device uses INTx messages for interrupt signaling. The PCI device uses *INTx#* signals for signaling interrupts to the PCIe-to-PCI bridge, and this bridge communicates via INTA assertion messages with the interrupt controller.

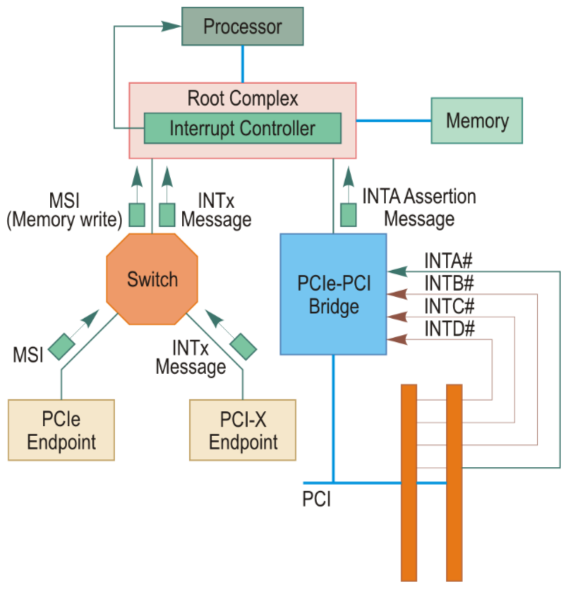

**Figure 2.7.** Interrupt signaling methods in a PCIe system.

## 2.4. Physical Layer

The physical layer performs the transmission on a PCIe link of packets received from the data link layer. Also, the physical layer receives packets from a PCIe link and sends them to the data link layer.

The physical layer is divided into two portions, the logical physical layer and the electrical physical layer. The logical physical layer contains digital logic for processing packets before transmitting them on a PCIe link and for processing packets received from a link before sending them to the data link layer. The electrical physical layer represents the analog interface used for connecting to the PCIe link. This layer consists of differential drivers and receivers for each lane of a PCIe link.

### 2.4.1. Transmit Section

Packets generated by the transaction layer or the data link layer are received by the physical layer and stored into a buffer. These packets are then framed with a *Start* character and an *End* character of one byte each. These characters are used by a receiver device to detect the start and end of a packet.

When a packet is sent on a PCIe link containing several communication lanes, the packet bytes are sent in an interleaved manner, which means that successive bytes of the packet are sent on successive communication lanes of the PCIe link. Although this data interleaving requires a significant hardware complexity for correctly assemble the bytes received in a packet, this method significantly reduces the delay with which a particular byte will be received on a link.

Each byte of a packet is then scrambled using a linear feedback shift register. By this operation, repeated bit patterns are eliminated from the transmitted data stream, with the aim of reducing the electromagnetic interferences. The resultant bytes are then encoded through a method that ensures to limit the length of streams with successive 1 and 0 bits. The main purpose of this encoding is to create sufficient 1-to-0 and 0-to-1 transitions in the bit stream transmitted, which will facilitate the recreation of a receive clock signal by the receiver device with the aid of a PLL (*Phase-Locked Loop*) circuitry. Hence, there is no need to send a clock signal for synchronization along the data.

In the first versions of the PCIe bus (up to version 3.0), the encoding method used is 8b/10b, by which each byte is encoded into a 10-bit symbol. With this method, the effective bandwidth is reduced by 20%. In version 3.0 of the PCIe bus the 128b/130b encoding method is used, by which the effective bandwidth is reduced by only about 1.5%.

The encoded bytes of a packet are then converted into a serial bit stream using a parallel-to-serial converter and are sent on the communication lanes of the PCIe link.

### 2.4.2. Receive Section

The receive section of the physical layer collects the serial bit streams arriving on each communication lane of the link. The bit streams are converted into 10-bit or 130-bit symbols using a serial-to-parallel converter. The receiver logic also contains a buffer which compensates for the variation between the transmitter clock frequency and the receiver clock frequency.

The 10-bit or 130-bit symbols are converted into bytes with a decoder. The bytes are then descrambled, and the assembly logic recreates the original packet transmitted by combining the bytes received on the communication lanes of the PCIe link.

### 2.4.3. Link Initialization and Training

An additional function of the physical layer is the process of PCIe link initialization and training. This process is automatic and does not involve the software layer. During the link initialization and training process several operational parameters are determined, such as link width, link data rate, communication lane reversal, polarity inversion, and signal skew compensation within a multi-lane link.

*Link width*. It is possible to connect two devices via ports with a different number of communication lanes per link. After initialization, the link width will be set to the minimum lane width of the two connected ports. For instance, a certain device with an x2 PCIe port may be connected to another device with an x4 PCIe port. For communication between two devices, the link width will be set to x2.

*Link data rate*. Initially, a link's data rate is set to the minimum value of 2.5 Gbits/s. During link training, each device advertises its highest data rate that is capable of. The link will be initialized with the highest common frequency supported by the two devices at opposite ends of the link.

*Communication lane reversal*. When a link contains several communication lanes, these are numbered. When two devices are physically connected, the communication lanes of the devices' ports may not be connected correctly. In such a case, link training allows for the lane numbers to be reversed, so that the lane numbers of adjacent ports on each end of the link correspond.

*Polarity inversion*. The D+ and D- differential wire pairs of two devices may not be connected correctly. In this case, as the result of link training, the receiver device will invert the polarity of the receiver circuit's terminals.

*Skew compensation*. In the case of a multi-lane link, due to length variations of physical wires and different characteristics of driver/receiver circuitry, bit streams on a lane may be received skewed with respect to other lanes. The receiver circuits must compensate for this skew by adding delays on some lanes.

## 2.5. Versions of the PCI Express Specification

The first version 1.0a of the PCIe bus specification has been released by the PCI-SIG organization in 2003. This version specified an operating frequency of 2.5 GHz and a maximum data bandwidth of 250 MB/s per communication lane.

Version 1.1 has been released in 2005. This version introduced several bus enhancements but did not specify higher data bandwidths.

Versions 2.0 and 2.1 have been released in 2007. The operating frequency has been doubled to 5 MHz, which allows a maximum data bandwidth of 500 MB/s per communication lane. Therefore, an x32 PCIe link may provide a maximum theoretical data bandwidth of 16 GB/s. The 2.x PCIe specifications introduce enhancements of the point-to-point transfer protocol and of the software layer architecture. The connectors of PCIe 2.x motherboards are compatible with PCIe 1.x expansion cards. In general, PCIe 2.x expansion cards are compatible with PCIe 1.x motherboards, operating at their lower frequency.

Version 3.0 of the PCIe specification has been released by the PCI-SIG organization in 2010. The operating frequency has been increased to 8 GHz, with a maximum data bandwidth of 8 GT/s (Giga Transfers per second) or 985 MB/s per communication lane. The specification introduced improvements related to the signaling protocol, data integrity, and error recovery. Also, it introduced the 128b/130b data encoding, which is more efficient than the 8b/10b encoding used by the previous PCIe versions.

Version 4.0, released in 2017, doubled the data bandwidth provided by version 3.0 to 16 GT/s or 1.97 GB/s per communication lane. Power consumption of devices in their active and inactive state has been optimized. This version also introduced the OCuLink-2 connector, which is the second version of the OCuLink (*Optical-Copper Link*) connector, supporting up to four lanes over copper wires; a fiber optic version may be developed in the future.

Version 5.0 has been released in May 2019. This version doubles the maximum data bandwidth of the previous version again to 32 GT/s per communication lane, and it maintains backwards compatibility with the previous versions.

Version 6.0 has been released in January 2022. With this version, the maximum data bandwidth may reach 64 GT/s per communication lane, so that an x32 PCIe link may provide a maximum theoretical bandwidth of 256 GB/s.

## 2.6. Configuration Registers

Each PCIe function (device) implements a set of configuration registers that enable the software layer to discover the existence of a function and to configure it for normal operation. At the request of application software, the root complex of the PCIe system initiates configuration transactions for reading from or writing to configuration registers of PCIe functions.

### 2.6.1. PCIe Function Configuration Space

A PCIe function's configuration registers are implemented in the configuration space of the PCIe architecture. Unlike a PCI or PCI-X function, which may have a configuration space of 256 B, a PCIe function has a configuration space extended to 4 KB. It follows that the size of PCIe configuration space is 256 MB. This is obtained by multiplying the size of 4 KB by 8 functions for a device, by 32 devices on a bus, and by 256 buses in a PCIe system.

The structure of a PCIe function's configuration space is illustrated in Figure 2.8.

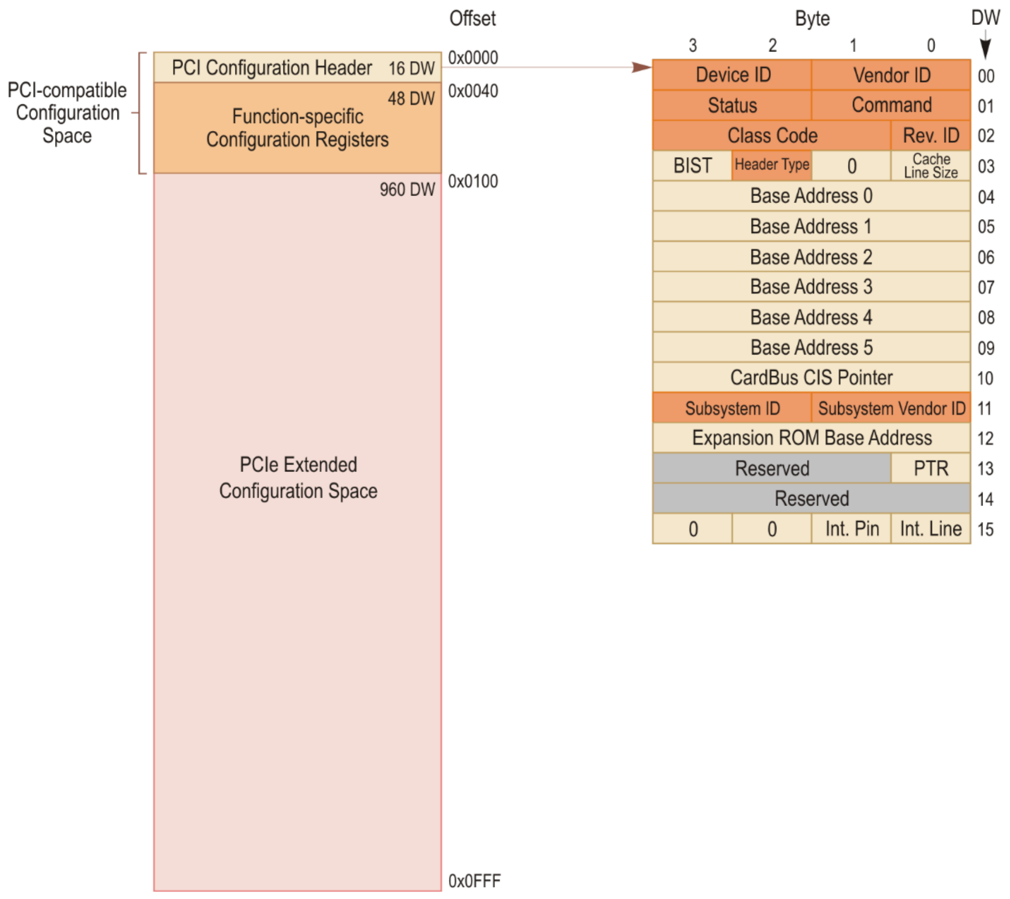

**Figure 2.8.** Structure of a PCIe function's configuration space.

A PCIe function's configuration space is divided into two sections. The first section represents the PCI-compatible configuration space, and it occupies the first 256 B (64 double-words of 32 bits) of the 4-KB space. The first 16 double-words of this section represent the PCI configuration header, while the remaining 48 double-words are reserved for the implementation of function-specific configuration registers.

The second section of a PCIe function's configuration space represents the PCIe extended configuration space and it occupies 3840 B (960 double-words). This space is used to implement the PCIe extended capability registers, which are optional. Examples of such registers are the advanced error reporting capability register set, virtual channel capability register set, and device serial number capability register set.

The PCI-compatible configuration space may be accessed via two methods, either through the PCI-compatible configuration mechanism or the PCIe enhanced configuration mechanism. These access mechanisms are presented in the following sections. A PCIe function's extended configuration space can only be accessed through the PCIe enhanced configuration mechanism.

### 2.6.2. PCI-Compatible Configuration Mechanism

For PC-AT compatible systems based on x86 processors, the PCI specification (version 2.3) defines a method that uses I/O accesses to request the host-PCI bridge to perform configuration transactions in order to access the configuration registers of PCI functions. The specification does not define a configuration mechanism for other systems that are not PC-AT compatible.

Due to the limitation of I/O space for x86 processors to 64 KB, the configuration registers cannot be mapped directly into the processor's I/O space. Access to these registers may be performed indirectly via two 32-bit I/O ports implemented in the host-PCI bridge (for the PCIe bus, this bridge is located in the root complex). These ports are the following:

- The configuration address port, with the I/O address of 0x0CF8.
- The configuration data port, with the I/O address of 0x0CFC.

Access to one of a PCIe function's PCI-compatible configuration registers is performed in two steps:

1. Write to the configuration address port the PCI bus number, device number, function number, and configuration register address (double-word number), in the format illustrated in Figure 2.9, and set the *Enable* bit (bit 31) of this port to one.
2. Perform a read from or write to the configuration data port of a double-word (32 bits).

In response to these operations, the host-PCI bridge within the root complex compares the specified bus number to the numbers of the buses connected to that bridge and, if the specified bus is connected to the bridge, it initiates a configuration read or write transaction (based on whether the processor is performing a read or write operation with the configuration data port).

The configuration address port only latches information written to this port when the processor performs a 32-bit write to the port. Therefore, latching the information is not possible through several 8-bit or 16-bit write operations to the port. A 32-bit read from the port returns its contents. The information written to the configuration address port must be organized in the manner presented in Figure 2.9.

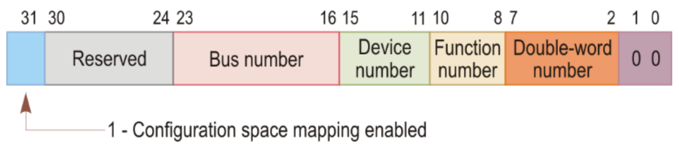

**Figure 2.9.** Structure of information written to the configuration address port.

The meaning of the fields in the configuration address port is presented next.

- Bit 31 represents the enable bit for mapping the configuration space. This bit must be set to one to enable the translation of a subsequent processor access to the configuration data port into a transaction for accessing the configuration space. If this bit is set to zero and the processor initiates an access to the configuration data port, the operation will be translated into a transaction for accessing the I/O space and not the configuration space.
- Bits 30..24 are reserved and must be set to zero.
- Bits 23..16 identify the PCI bus number (0..255).

- Bits 15..11 identify the device number (0..31).
- Bits 10..8 identify the function number (0..7) within the device.
- Bits 7..2 identify the function's configuration register by the double-word number (0..63) within the function's PCI-compatible configuration space.
- Bits 1..0 are set to zero and cannot be changed.

### 2.6.3. PCIe Enhanced Configuration Mechanism

The PCIe enhanced configuration mechanism performs the mapping of PCIe architecture's configuration space to the main memory's address space. Each PCIe controller in a system is allocated an area in main memory. At system initialization, the BIOS determines the base address of the area in main memory allocated to a PCIe controller and communicates it to the root complex and operating system. The communication method is implementation specific and is not defined in the PCIe specification.

Each configuration register of a PCIe function is assigned a particular memory address in the area allocated to the PCIe controller of that function. The root complex of the PCIe system monitors the memory accesses, and if it detects an access to the area of 256 MB allocated to the PCIe controller, initiates a configuration transaction for accessing the PCIe configuration space.

The configuration space of each PCIe function starts at a 4-KB aligned address within the memory area of 256 MB. The structure of the address that has to be specified for accessing a particular byte, word, or double-word within the configuration space of a PCIe function is the following:

- Bits 63..28 represent the base address of the 256 MB memory area allocated as configuration space of the PCIe controller corresponding to the PCIe function.
- Bits 27..20 select the PCI bus (0..255).
- Bits 19..15 select the device (0..31).
- Bits 14..12 select the function (0..7) within the device.
- Bits 11..2 select the double-word (0..1023) within the PCIe function's configuration space.
- Bits 1..0 indicate the byte offset (0..3) within the selected double-word.

To access the extended configuration space of a PCIe function, it is necessary to determine the base address of the memory area allocated as configuration space for the PCIe functions of a PCIe controller. There are several methods that can be used to determine this base address. For instance, one of the methods is based on searching in memory for certain system tables and accessing these tables. These methods are not presented in this laboratory work.

#### Note

- The PciBaseAddressUEFI-e.cpp source file, available on the laboratory web page, contains the `PciBaseAddressUEFI()` function, which returns the base address of the PCIe extended configuration space as a 64-bit value (`DWORD64`).

### 2.6.4. Mandatory Configuration Header Registers

As presented in Section 2.6.1, the first 16 double-words of a device's PCI-compatible configuration space represent the PCI configuration header. The PCI specifications define three header formats, referred to as header type zero, one, and two. Header type two is defined for CardBus bridges, header type one is defined for PCI-to-PCI bridges, and header type zero is defined for all other devices. In this laboratory work, we only refer to header type zero.

The structure of configuration header type zero is illustrated in Figure 2.8. Part of the registers in this header must be implemented in every PCI or PCIe device, including bridges. These mandatory registers are shown in a darker color in Figure 2.8. The mandatory configuration header registers are described next.

### Note

- The PCI-e.h header file, which is used for the applications, defines the configuration header type zero in a structure called `PCI_CONFIG0`.

### Vendor ID Register

This 16-bit register identifies the manufacturer of the device. The vendor identifier is assigned by the PCI SIG organization. The value 0xFFFF is reserved and is returned by the host-PCI bridge when an attempt is made to perform a read from a configuration register in a non-existent PCI function.

### Device ID Register

This 16-bit register contains an identifier assigned by the device manufacturer that identifies the type of device. In conjunction with the Vendor ID register and possibly the Revision ID register, the Device ID register can be used to locate a function-specific driver for the device.

### Revision ID Register

This 8-bit register contains an identifier that is assigned by the device manufacturer and identifies the revision number of the device.

### Class Code Register

The structure of this register is illustrated in Figure 2.10. It is a 24-bit register divided into three 8-bit fields: class code (the upper byte), sub-class code (the middle byte), and programming interface (the lower byte). This register identifies the basic function of the device (for instance, a mass storage controller), a more specific device sub-class (such as SATA mass storage controller), and, in some cases, a register-specific programming interface.

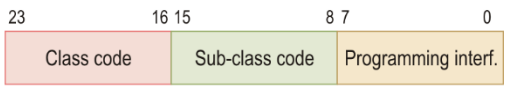

**Figure 2.10.** Structure of Class Code register.

For many class code/sub-class code combinations, the programming interface byte returns zero, and therefore it has no meaning. For other combinations, however, the programming interface byte does have meaning, as it identifies the exact register set layout of the function, which can vary from one implementation to another. For instance, there are different types of USB controllers with the same class code and sub-class code, but with different programming interfaces (e.g., UHCI, OHCI, EHCI, and XHCI).

### Note

- The PCI-e.h header file contains the currently defined class codes, sub-class codes, and programming interfaces, in a structure called `PCI_CLASS_TABLE`. This structure also contains pointers to two descriptors (texts) that can be used for decoding the information contained in the Class Code register: the first is a class and sub-class descriptor, and the second is a programming interface descriptor.

### Command Register

This 16-bit register provides basic control over the device's ability to perform PCI or PCIe transactions. It contains bits that allow to enable or disable the I/O address space decoder, enable or disable the memory address space decoder, enable or disable the function's ability to issue memory access requests or I/O requests, enable or disable the reporting of errors detected by the function, and enable or disable the function's ability to generate INTx interrupt messages. The Command register is not described in detail in this laboratory work.

#### Status Register

This 16-bit register traces the status of events related to the PCI or PCIe bus. It contains bits that indicate the interrupt status (whether the function has an interrupt request outstanding), whether a parity error has been detected, or whether a transaction has been aborted by the target or the initiator device. The Status register is not described in detail in this laboratory work.

Some bits of this register have RO (Read Only) attribute, while others have R/W (Read/Write) attribute. A particular feature of the bits that can be written is that they can be cleared, but not set. A bit can be cleared by writing a one to it; this attribute is denoted as RW1C (Read/Write 1 to Clear). This method was chosen to simplify programming. After reading the status and identifying the error bits that are set, the programmer can clear these bits by writing the value that was read back to the register.

#### Header Type Register

Bits 6..0 of this 8-bit register define the configuration header type. The following header types are currently defined:

0: Non-bridge function;
1: PCI(X)-to-PCI(X) bridge;
2: CardBus bridge.

Bit 7 defines the device as a single-function device (if bit 7 is 0) or a multi-function device (if bit 7 is 1). During configuration, the programmer may test the state of this bit to determine whether there are any other functions of the device that require configuration.

#### Subsystem Vendor ID and Subsystem ID Registers

The 16-bit Subsystem Vendor ID register contains an identifier assigned by the PCI SIG organization. The 16-bit Subsystem ID register contains an identifier assigned by the function vendor. A value of zero read from these registers indicates that there is no subsystem vendor ID and subsystem ID associated with the function.

A function may reside on an expansion card or within an embedded device. Functions that are designed around the same PCI, PCI-X, or PCIe core logic may have the same vendor ID and device ID assigned by the core logic vendor. In this case, the operating system would not be able to identify the correct driver to be used for that function. The Subsystem Vendor ID and Subsystem ID registers are used to uniquely identify the expansion card or subsystem that the function resides within. Consequently, the operating system can distinguish the difference between cards or subsystems manufactured by different vendors but designed around the same core logic.

### 2.6.5. Optional Configuration Header Registers

The most important optional configuration header registers are described next.

#### BIST Register

This register may be implemented by both an initiator and a target function. If a function supports a *Built-In Self-Test* (BIST) operation, it must implement this register, with the structure illustrated in Figure 2.11. If bit 7 of the BIST register is one, it means that the function supports a BIST operation. If the function does not support a BIST operation, this register will return the zero value when read. The function's BIST operation is invoked by setting bit 6 to one. The function should complete the BIST operation in a time limit of two seconds, and then it should reset bit 6. The test result is indicated in bits 3..0 of the register. A completion code of zero indicates successful completion of the test. A non-zero value represents a function-specific error code.

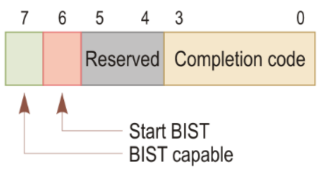

**Figure 2.11.** Structure of BIST register.

## Base Address Registers

Most functions use a memory address space and/or an I/O address space to implement a function-specific register set, which is used to control the function and identify its status. On power-up, the system must be configured such that each function's memory and I/O spaces occupy mutually exclusive address ranges. Therefore, the system must be able to detect what memory and I/O spaces are required by a certain function. In addition, the system must be able to program the function's address decoders in order to assign non-conflicting address ranges to them.

The Base Address Registers (BARs) are located in double-words 4..9 of the header space and they are used to implement a function's programmable memory and/or I/O decoders. Each register is 32-bits wide or 64-bits wide (in the case of a memory decoder whose associated memory block can be located above the 4 GB space). Bit 0 is a read-only bit and indicates whether the register is a memory decoder or an I/O decoder:

Bit 0 = 0: the register is a memory address decoder;
Bit 0 = 1: the register is an I/O address decoder.

Decoders may be implemented in any of the Base Address Registers. During configuration, the configuration software must check all six Base Address Registers in a function's configuration header to determine which registers are actually implemented.

## Structure of a Memory Base Address Register

A memory Base Address register might have a size of 32 bits or 64 bits. Figure 2.12 illustrates the structure of a 64-bit memory Base Address register. Bit 0 is zero and indicates a memory address decoder. Bits 2..1 define the size of the memory decoder:

00: 32-bit memory decoder;
10: 64-bit memory decoder.

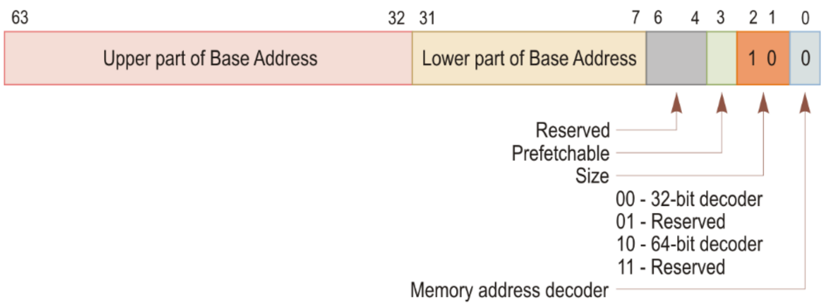

**Figure 2.12.** Structure of a 64-bit memory Base Address register.

For a 32-bit memory decoder, the Base Address register contains a start memory address in the first 4 GB of the memory address space. For a 64-bit memory decoder, the Base Address register contains a start address anywhere in the memory address space of $2^{64}$ bytes. In this case, the Base Address register occupies two consecutive double-words in the configuration header space. The first double-word contains the lower 32 bits of the memory start address and the second double-word contains the upper 32 bits of the memory start address.

Bit 3 indicates whether the memory block is prefetchable (bit 3 = 1) or not (bit 3 = 0). For a prefetchable memory block, it is acceptable for a bridge that resides between an initiator and a memory target to prefetch data from memory into a buffer in order to yield better performance.

Bits 31..7 for a 32-bit memory decoder and bits 63..7 for a 64-bit memory decoder contain the memory base address.

For each memory Base Address register, the configuration software should determine whether the register is implemented, what is the size of the register (32 bits or 64 bits), and what is the size of the memory space corresponding to the register. The size of the memory space can be determined using the following procedure:

1. Read the contents of the Base Address register into a temporary variable.

2. Write the value consisting of all one bits to the Base Address register.

3. Read back the contents of the Base Address register and then restore its contents from the temporary variable. If the value read is zero, it indicates that the Base Address register is not implemented, and the procedure is completed.

4. If the value read is not zero, scan the bits of the value upwards starting with the least significant bit of the Base Address field (bit 7) until the first bit set to one is found. The binary-weighted value of the least significant bit set to one represents the size of the memory space associated with the Base Address register.

As an example, assume that the value 0xFFFFFFFF is written to a Base Address register and the value read back from the register is 0xFFF00000. As the value read back is not zero, the register is implemented. Since bit 0 is zero and bits 2..1 are 00, the register is a 32-bit memory address decoder. Bit 20 is the first bit set to one in the Base Address field. The binary-weighted value of this bit is $2^{20}$, which means that the size of the memory space corresponding to this register is 1 MB.

### Structure of an I/O Base Address Register

An I/O Base Address register has a size of 32 bits. Figure 2.13 illustrates the structure of an I/O Base Address register. Bit 0 is one and indicates an I/O address decoder. Bit 1 is reserved and always returns zero when read. Bits 31..2 represent the Base Address field.

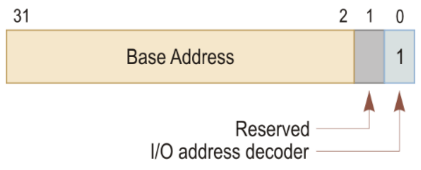

**Figure 2.13.** Structure of an I/O Base Address register.

The upper 16 bits of an I/O Base Address register may be hardwired to zero by the manufacturer when a function is designed specifically for a PC-compatible computer, since Intel x86 processors are limited to an I/O space of 64 KB.

The size of the I/O space corresponding to an I/O Base Address register can be determined using the same procedure used for determining the size of the memory space. The only difference is that the least significant bit of the Base Address field is bit 2 instead of bit 7. As a second example, assume that the value 0xFFFFFFFF is written to a Base Address register and the value read back is 0xFFFFFF01. Bit 0 is one, indicating that the register is an I/O address decoder. Scanning upwards starting with bit 2, bit 8 is the first bit set to one in the Base Address field. The binary-weighted value of this bit is $2^8$, which means that the size of the I/O space corresponding to this register is 256 bytes.

## 2.7. Applications

**2.7.1.** Answer the following questions:

a. What are the improvements introduced by the PCIe bus compared to the previous PCI and PCI-X buses?

b. What are the main components of the PCIe bus topology?

c. What are the methods that can be used by PCIe devices for interrupt signaling?

d. What are the operational parameters determined during link initialization and training?

e. What are the configuration registers that must be used to uniquely identify a PCIe function?

**2.7.2.** Create a Windows application for identifying each PCIe device in the computer. As template for the Windows application, use the AppScroll-e application, whose source files are available on the laboratory web page in the AppScroll-e.zip archive. Perform the following operations to create the application project:

1. In the *Visual Studio 2022* programming environment, create a new empty *Windows Desktop* project with the *Windows Desktop Wizard*. Check the *Place solution and project in the same directory* option to avoid creating another folder for the solution.
2. Verify that the active solution platform is set to x64.
3. Change the *Character Set* project property by opening the *Properties* dialog window. In this window, expand the *Configuration Properties* option, expand the *Advanced* option, select the *Character Set* line in the right tab, and choose the *Not Set* option.
4. Copy to the project folder the files contained in the AppScroll-e.zip archive and add these files to the project.
5. Copy to the project folder the Hw.h and Hw64.lib files from the folder of a previously created project.
6. Copy to the project folder the header files and source file from the PCI-e.zip archive, available on the laboratory web page.
7. Add to the project the Hw.h, PCI-e.h, Pci-vendor-dev.h, and PciBaseAddressUEFI-e.cpp files.
8. Specify the Hw64.lib file as an additional dependency for the linker.
9. In the AppScroll-e.cpp source file, add `#include` directives to include the PCI-e.h and Pci-vendor-dev.h header files. Declare `PciBaseAddressUEFI()` as a function that has no parameters and returns a `DWORD64` value.

In the AppScroll-e.cpp source file, first call the `PciBaseAddressUEFI()` function to determine the base address of the PCIe extended configuration space and store the base address in a global variable. If the function returns 0, the base address cannot be successfully determined, and in this case the application should return with an error code. Otherwise, display the base address as two double-words. Next, write a function that returns a pointer to a PCIe function's configuration header using the PCIe enhanced configuration mechanism. The function has as input parameters the bus number, device number, and PCIe function number, and it returns a pointer to a `PCI_CONFIG0` structure containing the PCIe function's configuration header. The enhanced configuration mechanism is described in Section 2.6.3. In this function, use the global variable containing the base address of the PCIe extended configuration space. Finally, use this function to search for PCIe devices on each bus between 0 and 63, for each device (0..31), and for each function (0..7) of a device. For each existing PCIe device, the following information should be displayed (on separate lines):

- Bus number, device number, function number;
- Class code, sub-class code, programming interface, subsystem vendor ID, subsystem ID;
- Class/sub-class descriptor, programming interface descriptor.

Use the structures defined in the PCI-e.h header file. For displaying the class/sub-class descriptor and the programming interface descriptor, search in the `PciClassTable` array using the class code, sub-class code, and programming interface as search keys.

**Notes**

- If the Vendor ID register of a PCIe function returns the value 0xFFFF when read, it means that the function does not exist. In this case, no message should be displayed since there are many non-existing functions.
- The configuration registers of a PCIe function should not be accessed directly, but rather via the Marvin HW driver. For example, assuming that `pRegPci` is a pointer to a function's configuration header, the Vendor ID register can be read into the `wVendorID` variable as follows:

```
wVendorID = _inmw((DWORD_PTR)&pRegPci->VendorID);
```

**2.7.3.** Extend Application 2.7.2 to display additional information about the existing PCIe devices in the computer. The additional information that should be displayed is the following:

- Vendor ID, vendor descriptor;
- Device ID, chip descriptors.

Use the PCI-vendor-dev.h header file that has been added to the project. To display the vendor descriptor, search in the `PciVenTable` array using the vendor ID as search key and display the `CONST CHAR *VenFull` member of the `PCI_VENTABLE` structure. To display the chip descriptors, search in the `PciDevTable` array using the vendor ID and device ID as search keys and display the `CONST CHAR *Chip` and `CONST CHAR *ChipDesc` members of the `PCI_DEVTABLE` structure.

**Notes**

- The number of entries in the `PciVenTable` array is defined as `PCI_VENTABLE_LEN`.
- The number of entries in the `PciDevTable` array is defined as `PCI_DEVTABLE_LEN`.
- When a device ID cannot be found in the `PciDevTable` array, some information about that device might be found in the pci.ids file from *The PCI ID Repository* website (http://pci-ids.ucw.cz/), also available on the laboratory web page.

**2.7.4.** Extend Application 2.7.3 to display the same information specified in Application 2.7.2 about the PCIe devices, but this time by using the PCI-compatible configuration mechanism for accessing the configuration space. This mechanism is described in Section 2.6.2. First, write a function that reads the contents of a single double-word from a PCIe function's configuration header using the PCI-compatible configuration mechanism. The function has the following input parameters: bus number, device number, PCIe function number, and double-word number. The function returns the contents of the specified double-word. Then, use this function to retrieve the contents of the registers that were used for Application 2.7.2 by specifying the appropriate double-word number when calling the function (the double-word numbers are shown in Figure 2.8). Extract the relevant information from these registers (e.g., class code, sub-class code, programming interface) and display the same information as displayed for Application 2.7.2.

**2.7.5.** Create a Windows application to identify the SMBus (*System Management Bus*) controller of the computer and to display the contents of its base address registers. After creating the project, write a function that searches for a PCIe device with the class code and sub-class code specified as parameters. The function returns in a double-word the bus number, device number, and function number of the PCIe device; it returns 0 if the PCIe device with the specified class code and sub-class code cannot be found. The function searches for the specified PCIe device on each bus between 0 and 63, each device (0..31), and each PCIe function (0..7) of a device. The function uses the PCIe enhanced configuration mechanism for accessing the configuration space. Use this function to identify the SMBus controller, which has a class code of 0x0C and sub-class code of 0x05. After identifying the controller, display its bus number, device number, and function number. Then, for each of the six base address registers of the controller, perform the following operations:

- Display the register type (memory decoder or I/O decoder);
- If the register is a memory decoder, display its size (32 bits or 64 bits) and the corresponding memory base address;
- If the register is an I/O decoder, display the corresponding I/O base address.

## Bibliography

[1] Ajanovic, J., "PCI Express (PCIe) 3.0 Accelerator Features", Intel Corporation, 2008, http://www.intel.com/content/dam/doc/white-paper/pci-express3-accelerator-white-paper.pdf.

[2] Bhatt, A. V., "Creating a PCI Express Interconnect", Intel Corporation, 2002, http://www.advancedaudiorentals.com/phpkb/admin/attachments/pci_express_white_paper.pdf.

[3] Budruk, R., Anderson, D., Shanley, T., *PCI Express System Architecture*, MindShare Inc., Addison-Wesley Developer's Press, 2008, https://www.mindshare.com/files/ebooks/PCI%20Express%20System%20Architecture.pdf.

[4] PCI-SIG, "PCI Code and ID Assignment Specification", Revision 1.11, January 24, 2019.

[5] PCI-SIG, "PCI Express Base Specification Revision 3.0", November 10, 2010.

[6] Shanley, T., Anderson, D., *PCI System Architecture*, Fourth Edition, MindShare Inc., Addison-Wesley Developer's Press, 1999.

[7] PCI Vendor and Device Lists, http://www.pcidatabase.com.

[8] The PCI ID Repository, https://pci-ids.ucw.cz.