4. UNIVERSAL SERIAL BUS

This laboratory work presents the Universal Serial Bus and the interface based on this bus. The following topics are presented: basic concepts of the USB, connectors and cables used, transfer types, communication model, bus protocol, packet structure, USB descriptors, and USB commands. In addition, the laboratory work introduces the USB HID class and describes how the communication between the computer and devices from this class can be performed.

4.1. Overview of USB

The development of the *Universal Serial Bus* (USB) started in 1995 by a group of companies which included Compaq, Digital, IBM, Intel, Microsoft, NEC, and Northern Telecom. These companies have later joined into the *USB Implementers Forum* (USB-IF, http://www.usb.org), which published the first version of the USB standard. This forum, which has been extended with a large number of companies, continues to update the USB standards for USB controllers and various categories of devices that can be connected to the USB.

One of the motivations for developing the USB was to simplify the interconnections between the computer and peripherals by reducing the number of cables that are connected to the computer and by using the same type of connector for various categories of peripherals. In a system containing a USB, the peripherals can be connected in series or in a star topology on several tiers, a single peripheral being connected to a USB port of the host computer. Another motivation for developing the USB was to provide a higher transfer rate than those allowed by the serial and parallel ports. Although with the first versions (1.0 and 1.1) of the USB the maximum transfer rate was only 12 Mbits/s, this rate has been increased up to 480 Mbits/s with version 2.0 of USB and to 5 Gbits/s or 10 Gbits/s with versions 3.0 and 3.1 of USB. Another aim was to offer the possibility for adding peripherals to the computer in a simple way, without opening its case, without switching off the power and without having to reboot the operating system.

The computer equipped with a USB port detects the attachment of a new peripheral and automatically determines the resources it needs, including the software driver and transfer rate. One of the computer's peripherals, e.g., the keyboard or the monitor, is attached to a USB port of the computer. Other peripherals may attach to a distributor (hub) located within the keyboard or monitor, being possible to make tree-like connections. Peripherals can be located at up to 5 m one to the other or to a hub. In total, up to 127 USB peripherals can be connected to a computer, and these are powered with a voltage of +5 V via the USB cable. All USB peripherals use a standard connector, eliminating the need to use different connectors for different types of peripherals.

The main components of a system that uses the USB are USB devices, USB cables, and the system software. The devices on the USB are physically connected to the host computer using a star topology on several tiers, as illustrated in Figure 4.1.

There are two categories of USB devices: hubs and functions. A *hub* represents a special category of USB device, which provides additional attachment points for other USB devices. These attachment points are referred to as ports. The host computer contains a root hub, which provides one or more attachment points. In addition, this hub contains the USB bus controller; each bus has a single bus controller.

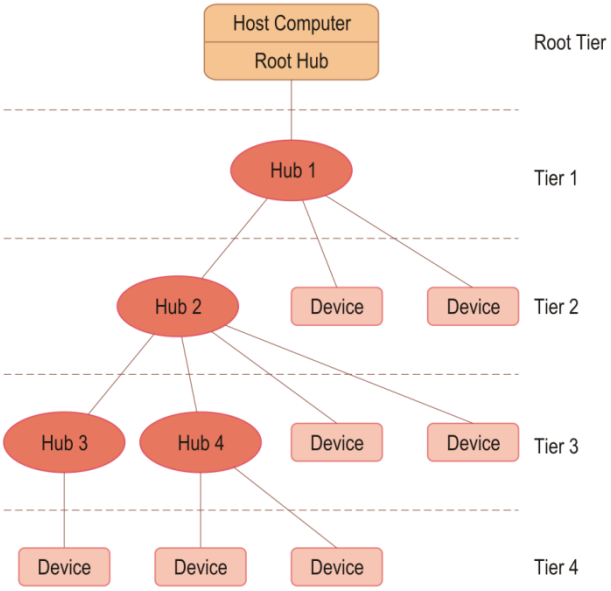


Figure 4.1. USB topology.

Figure 4.2 presents a typical USB hub. The upstream port of the hub allows connection to the host computer or to a hub on the higher topology tier. Each of the downstream ports allows connection to a hub or to a function on the lower tier. Hubs can be cascaded up to five levels. The hub detects dynamic attachment and detachment of a peripheral and provides a power of at least 0.5 W for each peripheral during initialization. Under control of the system software, the hub can provide an additional power for the operation of peripherals, up to 2.5 W, 4.5 W, or 9 W (depending on the USB version). Some peripherals, such as the keyboard or light pen, can be supplied only with the voltage provided by the bus cable, while others can have their own power supply.

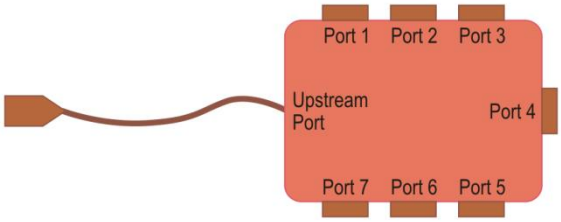


Figure 4.2. Ports of a typical USB hub.

A hub consists of two parts: a controller and a repeater. The controller contains interface registers for communication with the host computer. The status and control commands enable the host to configure the hub, to monitor and control its ports. The repeater is a protocol-controlled switch between the upstream port and downstream ports. Also, the repeater monitors the signals on the ports and manages the transactions that are addressed to it. All the other transactions are repeated to the attached devices. Each downstream port can be individually enabled and attached to either high-speed or low-speed devices. The low-speed ports are isolated from the high-speed signals.

Figure 4.3 illustrates how the USB hubs provide connectivity in a computer system.

A *function* is a USB device that is able to send or receive data or control information over the bus. This device must respond to transaction requests sent by the host computer. A function is usually implemented as a separate peripheral connected through a cable to a port of a hub. However, a single physical device may contain multiple functions. For instance, a keyboard and a tracking device may be combined into a single physical device. Within such a compound device, individual functions are attached to a hub, and this internal hub is connected to the USB.

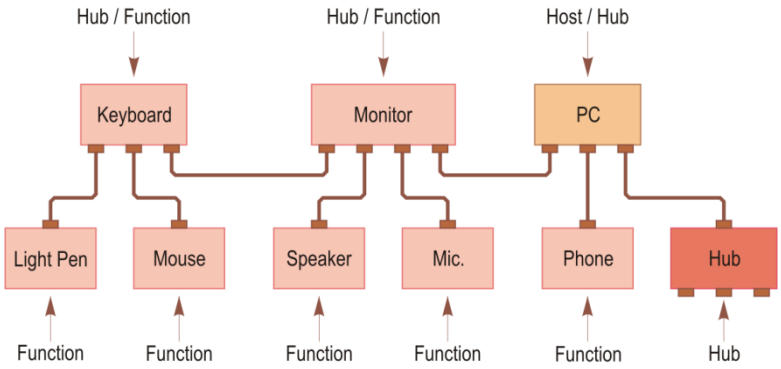


Figure 4.3. Using the USB hubs in a computer system.

Each function contains configuration information that describes its capabilities and resources required. This information is sent to the host computer as answer to a control transaction. Before using a function, it must be configured by the host computer. This configuration implies allocating USB bandwidth and selecting specific configuration options.

The system software ensures a uniform handling of the I/O system by all the application programs. By hiding hardware implementation details, the portability of application programs is ensured. The system software manages the dynamic attachment and removal of peripherals. The attachment phase, referred to as *enumeration*, implies communicating with a peripheral to determine the device driver that should be loaded and assigning a unique address to it. During operation, the host computer initiates transactions with some peripherals. The information is sent over the bus in the form of packages, which are received by every peripheral. The packages contain the address of the target peripheral; only that peripheral will accept a particular transaction and will answer accordingly.

The USB controller, located on the motherboard of the host computer, has its own specifications. With the fist USB versions, there were two specifications for these controllers. The first of these, *Universal Host Controller Interface* (UHCI), was developed by Intel and allowed simplifying the circuitry, the more complex part being the software. The second specification, *Open Host Controller Interface* (OHCI), was developed by Compaq, Microsoft, and National Semiconductor; this specification allowed simplifying the software, the more complex part being the hardware. With the introduction of USB version 2.0, developing a new specification for the bus controllers was required. This specification, called *Enhanced Host Controller Interface* (EHCI), was developed by several companies, such as Intel, Compaq, NEC, Microsoft, and Lucent Technologies. The *Extensible Host Controller Interface* (xHCI) has been defined by Intel with contribution from many companies. This specification supports all USB speeds, including the higher speeds enabled by USB version 3.0 and later versions, under a single protocol stack. Compared to the previous specifications, the xHCI specification improves the power efficiency, provides support for virtualization, and simplifies the driver architecture.

There are separate specifications for various categories (classes) of USB peripherals. A USB class represents a group of peripherals or interfaces with similar attributes or services. For instance, two peripherals or interfaces are included into the same class if they use data streams with the same format for communication with the host computer. Among the USB peripheral classes, the following are mentioned: audio devices, communication devices (modems, analog and digital phones, network adapters), human interface devices (HID), still-image capture devices (digital cameras), printers, mass storage devices, and video devices.

4.2. USB Versions

Version 1.0 of the USB standard was published in 1996, this being followed by version 1.1, adopted in 1998. The maximum transfer rate specified by these versions is 12 Mbits/s. This transfer rate is enough for peripherals such as telephones or digital speakers

(which contain a digital-to-analog converter). For slow peripherals, such as keyboards or light pens, a low-speed channel has been provided, with a transfer rate of 1.5 Mbits/s.

Version 2.0 of the USB standard was published in 2000. This version (also called *Hi-Speed* USB) allows increasing the transfer rate with a factor of 40 compared to version 1.1, from 12 Mbits/s up to 480 Mbits/s. This extension of the USB specifications allows using the same cables, connectors, and software drivers. However, users can benefit of an additional variety of peripherals, e.g., digital video cameras, scanners, printers, network adapters, hard disk drives, or optical disc drives. Figure 4.4 illustrates the USB 2.0 logo.

Figure 4.4. USB 2.0 logo.

USB 2.0 peripherals with higher transfer rates are connected to a USB 2.0 hub. A USB 2.0 hub accepts high-speed transactions and supplies the data with the rates corresponding to the USB 2.0 and USB 1.1 peripherals. The possibility to use high-speed transfers is negotiated with each peripheral, and if a peripheral does not support these transfers, the connection with this peripheral will operate at a lower speed of 12 Mbits/s or 1.5 Mbits/s. This involves a higher complexity of the hubs and requires storing temporarily the data received. A USB 2.0 hub has output ports for high-speed transfers and output ports for full-speed transfers.

The USB "On-The-Go" (OTG) specifications have been developed as a supplement to the USB 2.0 specifications with the aim to allow direct connections between mobile devices, without using a computer. The standard USB uses a master/slave architecture, where the host computer has the master role, and a peripheral has the slave role. Only the host computer can initiate data transfers over the bus. Peripherals can only respond to transfer requests initiated by the host computer. With the USB OTG version, a peripheral can have either the master role or the slave role. The master and slave roles can be exchanged dynamically during operation with a protocol called *Host Negotiation Protocol* (HNP). Thus, any peripheral compatible with the USB OTG specifications can initiate data transfers over the bus. The condition is that the two devices that communicate be connected directly and not through a hub.

An example of using the USB OTG version is that of a tablet computer or mobile phone, which can have the default slave role for a PC (for data synchronization) or the default master role for a printer. Another example is that of a printer, which can have the slave role for a tablet computer and the master role for a digital camera, if it allows reading files from the camera to print them. Figure 4.5 illustrates the USB OTG logo.

Figure 4.5. USB OTG logo.

The specifications of version 3.0 of the USB standard have been completed by the *USB 3.0 Promoter Group* in 2008 and have been transferred to the *USB Implementers Forum*. This version introduces the *SuperSpeed* bus, which allows a new transfer mode with the maximum speed of 5 Gbits/s. In the *SuperSpeed* mode, two simplex differential channels are used in addition to the existing differential channel for the conventional mode. The technology is similar to that of version 2.0 of the *PCI Express* bus. The same 8b/10b encoding is used, which allows a maximum transfer rate of 500 MB/s; it is possible to achieve a transfer rate of 400 MB/s if the data encoding method is considered. Figure 4.6 illustrates the USB *SuperSpeed* logo.

Figure 4.6. USB *SuperSpeed* logo.

The specifications of USB version 3.1 have been released in 2013. This version introduces a faster transfer mode called "*SuperSpeed*+ USB 10 Gbps", which increases the data rate up to 10 Gbits/s in the *USB 3.1 Gen 2* mode, double that of USB 3.0 (which has been renamed *USB 3.1 Gen 1*). The data encoding method has been changed from 8b/10b to 128b/132b, which reduces the encoding overhead to just 3%. The USB 3.1 specifications ensure compatibility with the previous versions USB 3.0 and USB 2.0. Figure 4.7 illustrates the USB *SuperSpeed*+ 10 Gbps logo.

Figure 4.7. USB *SuperSpeed*+ 10 Gbps logo.

The specifications of USB version 3.2, released in 2017, introduce a transfer mode called "*SuperSpeed*+ USB 20 Gbps" using existing USB 3.1 connectors and cables. Doubling of the data rate is achieved by operation on two lanes instead of on a single lane, using the existing wires intended to offer flipping capabilities to the Type-C connector (this connector is presented in Section 4.3). The same 128b/132b data encoding method is used as in version 3.1. Compatibility with the previous versions is ensured by using one of the following transfer modes: USB 3.2 Gen 1x1 (one lane, *SuperSpeed* at 5 Gbits/s), USB Gen 1x2 (two lanes, *SuperSpeed* at 10 Gbits/s), or USB Gen 2x1 (one lane, *SuperSpeed*+ at 10 Gbits/s).

The *USB Power Delivery* specification has been developed as an extension of the USB standards. This extension specifies using certified USB cables with standard USB connectors to deliver increased power to certain devices. Devices can request higher currents and supply voltages from host computers that comply with this specification, up to 2 A at 5 V (10 W). Optionally, the current consumption can be increased up to 3 A or 5 A at either 12 V (36 W or 60 W) or 20 V (60 W or 100 W). This allows laptop computers to be charged similarly with tablets and smartphones, via their USB ports, which may eliminate in the future the various proprietary charging ports.

4.3. Connectors and Cables

The original specifications of the USB defined two types of plugs, placed at the two ends of a USB cable, and two types of sockets, placed in a hub or peripheral. The plugs and sockets are designated as Type-A and Type-B.

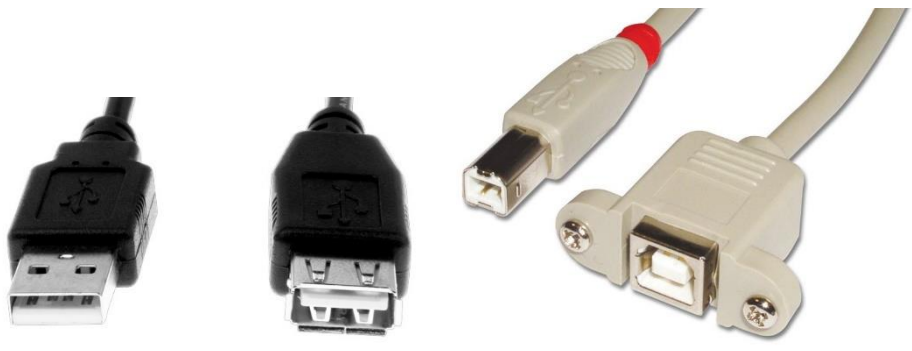


Figure 4.8. Type-A plug and socket (left); Type-B plug and socket (right).

Hubs (e.g., those of a host computer) contain a Type-A rectangular socket. Peripherals connect to this socket through a Type-A rectangular plug (Figure 4.8). Cables permanently attached to peripherals contain a Type-A plug. Usually though, peripherals are connected through a detachable cable. Peripherals contain a Type-B square socket, and the cable that connects the peripherals to a hub contains a Type-B plug at the end that connects to the peripheral and a Type-A plug at the end that connects to the hub. Therefore, it is not possible to incorrectly connect the cable.

The USB 2.0 specifications were modified after their release to include a Type-B plug and socket with smaller sizes. These connectors, called mini-B, contain five pins, and are used for mobile devices such as tablet computers, mobile phones, and digital cameras. These devices contain a mini-B socket, and the cables used for connecting these devices to a computer contain a mini-B plug at one end and a Type-A plug at the other end. The size of mini-B connectors is approximately 7x3 mm. Later, a mini-A connector was also defined. Figure 4.9 illustrates a mini-B plug near a Type-A plug.

Figure 4.9. Mini-B USB plug near a Type-A plug.

In 2007, Type-A and Type-B connectors with even smaller sizes were defined, called micro-A and micro-B. These connectors have the same width as the mini-B connectors, but the thickness is reduced to approximately half. The new connectors are intended to replace the mini-A and mini-B connectors, which are no longer used with new mobile devices.

The USB OTG specifications describe new socket types, mini-AB, and micro-AB. These sockets allow, through an adequate mechanical design, to connect either a Type-A plug or a Type-B plug with the appropriate size. The type of plug inserted is detected by means of an additional pin ID, which is grounded in a Type-A connector and is unconnected in a Type-B connector. When a Type-A connector is inserted into an AB socket, the socket will supply voltage to the cable and the device with the AB socket will have the master role. When a Type-B connector is inserted into an AB socket, the socket will be powered with the cable voltage and the device with the AB socket will have the slave role. The USB OTG specifications also describe various cable types that use small-size connectors or a combination between a small-size connector and a regular-size one. Figure 4.10 illustrates a micro-AB socket and a micro-B socket.

Figure 4.10. Micro-AB and micro-B USB OTG sockets.

USB version 3.0 defines new Type-A, Type-B, and micro-B connectors. Figure 4.11 illustrates, from left to right, a Type-A plug, a Type-B plug, and a micro-B plug. USB 3.0 Type-A plugs have the same shape as USB 2.0 Type-A plugs and are compatible with USB 2.0 Type-A receptacles. However, the USB 3.0 plugs contain five additional pins used by the two additional differential channels. The USB 3.0 specifications require that the inner part of

the connectors be colored in blue (Pantone 300C). USB 3.0 Type-B plugs have a different shape to include five new pins corresponding to the five additional pins in the Type-A connectors. These plugs are not backwards compatible with USB 2.0 Type-B receptacles. A USB 3.0 micro-B plug consists of two connectors, a USB 2.0 micro-B connector and an additional connector attached next to it. The USB 3.0 micro-B connectors are mostly used for external hard drives.

Figure 4.11. Type-A, Type-B, and micro-B USB 3.0 plugs.

USB Type-C is a new connector whose specifications have been developed and published by the *USB Implementers Forum* in 2014; they have been updated in 2015 and 2017. The Type-C plug connects to both host computers and USB devices, and is intended to replace various Type-A and Type-B plugs. The Type-C connector is reversible and contains 24 pins (Figure 4.12), which include four power pins, four ground pins, four pins for two USB 2.0 differential channels (though only one channel is implemented in a Type-C cable), eight pins for four *SuperSpeed* channels, and two configuration pins for cable orientation detection.

Figure 4.12. USB Type-C connector pins.

The size of a Type-C port is 8.4 x 2.6 mm, so it can even be used for small devices. The connector supports a current of 1.5 A or 3 A at 5 V (7.5 W or 15 W), or it may comply with the *USB Power Delivery* specification to deliver a power of up to 100 W. If a device contains a Type-C port, it does not necessarily support USB version 3.1 or 3.0 (for instance, it may only support USB version 2.0) or the *USB Power Delivery* specification. To connect an older device to a host computer with a Type-C receptacle, an adapter or a cable is required with a Type-A or Type-B plug on one end, and a Type-C plug on the other end. Type-C connectors and cables are also used by version 3 of Intel's proprietary technology, Thunderbolt, and this version is compatible with USB 3.1. Therefore, the same device can use both USB 3.1 and Thunderbolt operating modes.

Figure 4.13 illustrates a Type-C plug next to a Type-A plug.

Figure 4.13. USB Type-C and Type-A plugs.

To carry the signals and the supply voltage, USB 2.0 uses a cable with four wires, illustrated in Figure 4.14. The differential data signals are transmitted on the *D+* and *D-* lines, consisting of two twisted wires. The clock signal is encoded along with the data. The encod-

ing method used is called NRZI (*Non-Return to Zero Inverted*). With this method, a bit of 1 is represented by no change in the voltage level, and a bit of 0 is represented by a change in the voltage level, without returning to the reference voltage (zero) between the encoded bits. Extra bits are inserted into the data sent to ensure sufficient signal transitions to guarantee correct synchronization. A bit of 0 is inserted after every six consecutive bits of 1 before the data are encoded, to force a transition in the data stream. Each data packet is preceded by a synchronization field to allow receivers to synchronize their receive clocks.

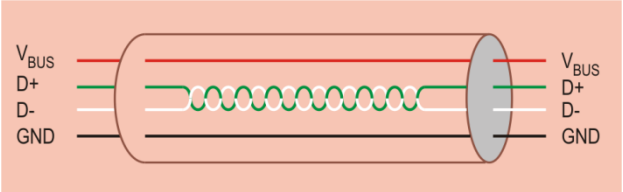


Figure 4.14. USB 2.0 cable.

A USB 2.0 cable also carries the supply voltage for the peripherals on the V_{BUS} and *GND* lines. The voltage on the V_{BUS} line is +5 V at source. To ensure correct voltage levels at peripheral inputs and adequate termination impedance, bus terminators are used at each end of the cable. Bus terminators also allow to detect attachment and removal of a peripheral and to differentiate between full-speed (12 Mbits/s) and low-speed (1.5 Mbits/s) peripherals. At the hub end of the cable, resistor terminators of 15 KΩ are used, through which the *D*+ and *D*– lines of the cable are connected to the ground. At the peripheral end of the cable, a resistor of 1.5 KΩ is used as terminator, through which one of the *D*+ or *D*– lines is connected to a voltage source between 3 V and 3.6 V. For a full-speed peripheral the resistor is connected to the *D*+ line, while for a low-speed device the resistor is connected to the *D*– line.

For simple identification of the wires within USB 2.0 cables, the standard specifies to use the colors indicated in Table 4.1 for these wires. The table also indicates the assignment of connector pins to the bus signals.

Table 4.1. Connector pin assignment to the USB 2.0 signals and the colors of cable wires.

Pin No.	Signal	Color
1	V_{BUS}	Red
2	D-	White
3	D+	Green
4	GND	Black

USB 3.0, USB 3.1, and USB 3.2 cables contain eight wires. Two wires are used for the power and ground signals, two wires are used for the differential channel of the conventional mode, and four wires are used for the two simplex differential channels of the *SuperSpeed* or *SuperSpeed+* mode.

Note

- It is not possible to interconnect two computers through a regular USB cable. Even if a cable with two Type-A connectors would be used, by interconnecting the computers two USB controllers would exist in a system, which is not allowed. However, there are special cables that contain a USB bridge as an integrated circuit, through which the communication between two host computers is possible.

4.4. Transfer Types

The USB architecture allows four types of data transfers: control, interrupt, bulk data, and isochronous.

Control transfers are used by the host computer drivers to configure devices that are attached to the system. Other drivers can use control transfers in implementation-specific ways.

Interrupt transfers are used for data that must be transferred with a limited delay. The transfer of such data may be requested by a device at any time, and the transfer rate over the bus cannot be lower than that specified by the device. Data for which interrupt transfers are used consist of event notification, characters, or coordinates that are organized as one or several bytes. An example is represented by the coordinates of a pointing device (mouse, graphics tablet). Interactive data may have certain limits of the response time that must be guaranteed by the bus.

Bulk data transfers are used with peripherals such as mass storage devices, printers, or scanners. Bulk data are sequential. The reliability of transfers is ensured at the hardware level by using an error detecting code and retrying an erroneous transfer a certain number of times. The transfer rate may vary depending on other bus activities.

Isochronous transfers (*isos* – equal, *chronos* – time) are used for data that must be delivered at a certain constant transfer rate and whose synchronization must be guaranteed. Isochronous has the meaning "with equal duration" or "that occurs at regular intervals". Isochronous data are generated in real time and must be delivered at the rate received to maintain their synchronization. In addition to the imposed transfer rate, for isochronous data the maximum delivery delay must also be ensured. The timely delivery of isochronous data is ensured at the expense of potential losses in the data stream. Hence, transmission errors are not corrected by hardware mechanisms, such as by their retransmission. In conclusion, isochronous transfers are characterized by timely data delivery and lack of retries in case of errors, because late data are no longer useful. Unlike isochronous transfers, asynchronous transfers are characterized by the fact that the reliability of data transmission is more important than ensuring synchronization. Therefore, retransmission of data is used in case of errors, even if delays occur for this reason.

A typical example of isochronous data is represented by video images. If the transfer rate of these data streams is not maintained, data losses will occur due to buffer capacity overruns. Even if the data are delivered by the USB at the appropriate rate, delays introduced by software may negatively affect applications that use these data, such video-conferencing applications.

Isochronous data streams are allocated a dedicated portion of USB bandwidth. The USB is also designed for minimal delay of isochronous data transfers.

4.5. USB Communication Model

A USB system allows the communication between an application program (client) running on a host computer and one or more USB devices attached to that computer. A USB physical device contains an interface with the bus, a USB logical device, and a function. A USB function can have different communication flow requirements for different interactions between that function and the host computer. By separating the different communication flows with a USB function, the USB system provides more efficient bus utilization. The communication flows use the bus to accomplish communication between the application program and the USB function. Each communication flow is terminated at an *endpoint* on a USB device.

An endpoint is a part of a USB device, representing the terminus of the communication flow between the host computer and device. Each USB logical device is composed of several independent endpoints. Each logical device has a unique address, which is assigned by the system when the device is attached to the bus. Each endpoint on a USB device is identified by a unique number, which is given at the device design time. Also, each endpoint has a certain direction of data flow: input (IN) for data transfers from the USB device to the host computer, or output (OUT) for data transfers from the host computer to the USB device. The combination of the device address, endpoint number, and direction allows the unique identification of each endpoint.

Endpoints contain input or output buffers through which the communication between the host computer and USB device is achieved. For instance, if the application program running on the host computer sends a packet addressed to a particular USB device, this packet

will be placed into the buffer of an output endpoint on the device. Later, the device controller will read the received packet from the buffer. If the device must send data to the host computer, it cannot place the data directly on the bus, because all the transactions are initiated by the host computer. The device will place the data into the buffer of an input endpoint, and these data will be transferred to the host computer when the host will send an input packet to request the transfer of data, if they are available.

Each device must contain at least the endpoints with number 0 (both the input and the output endpoint). These endpoints are used to transfer control and status information during the enumeration phase and as long as the device is operational and connected to the bus. Low-speed (1.5 Mbits/s) devices may contain only two more endpoints, besides those with number 0. Full-speed (12 Mbits/s) or high-speed devices may contain maximum 15 input endpoints and 15 output endpoints, besides the two endpoints with number 0.

A collection of endpoints on a USB device implements an *interface*. Such an interface indicates the USB device class, and this class will determine the generic device driver that will be used by the operating system for communication with the endpoints that implement the particular interface.

Communication between the application program on the host computer and an endpoint on a USB device is achieved through a logical connection called *pipe*. A pipe represents a link between a memory buffer on the host computer and an endpoint on the USB device. Each pipe is assigned some information, such as the required transfer rate, the transfer type, and the associated endpoint's characteristics, such as direction and buffer size.

There are two pipe communication modes: stream or message. In *stream mode*, data has no USB-defined structure. Data are transferred sequentially and have a predefined direction, input or output. Stream pipes support interrupt, bulk data, or isochronous transfers. These pipes are controlled either by the host computer or the USB device. In *message mode*, data have some USB-defined structure. Nevertheless, the content of the data transferred through a message pipe is not interpreted by the USB controller. These pipes are controlled by the host computer and only support control transfers, in both directions.

Communication between the input and output endpoints with number 0 and the host computer is achieved through a special pipe, called *default control pipe*. This message pipe is available immediately after the USB device is attached and reset, providing a bidirectional link to interrogate the device by the host computer's system software and to send the configuration information from the device to the host computer. After the USB device has been configured, other pipes required for the data transfers will be available; the default control pipe will be used afterwards by the host computer's system software for the control transfers.

4.6. USB Protocol

Similar to other more recent interfaces, the USB interface uses a packet-based protocol. All the transfers are initiated by the USB controller of the host computer. Bus transactions involve the transmission of four packet types:

- Token packet;
- Data packet;
- Handshake packet;
- Special packet.

Each transaction starts when the USB controller sends, based on a scheduling, a token packet that describes the type of transaction, its direction, the address of USB device, and the endpoint number. The transaction source then sends a data packet containing the data to be transferred, or it may indicate that it has no data to send by the fact that the data packet does not contain useful information. In general, the destination responds by a handshake packet indicating if the transfer has been completed successfully or if the endpoint is not available.

4.6.1. USB Packet Fields

The main fields of USB packets are described next.

Synchronization Field

All packets begin with a synchronization (SYNC) field, which is used by the receiver circuitry for synchronization with the transmitter clock. The synchronization field is 8 bits in length for full/low speed and 32 bits for high-speed. It contains a number of 6 or 30 successive transitions from 1 to 0 or vice versa, followed by a marker of two bits that is used to identify the end of the synchronization field.

Packet Identifier Field

The packet identifier (PID) field immediately follows the synchronization field. The PID field contains four bits that indicates the packet type followed by four check bits that ensures reliable decoding of the PID field. The check bits contain the 1's complement of the bits that represents the packet type. The PID field format is illustrated next, where $nPID_i$ represents the 1's complement of the PID_i bit.

7	6	5	4	3	2	1	0
$nPID_3$	$nPID_2$	$nPID_1$	$nPID_0$	PID_3	PID_2	PID_1	PID_0

The host computer and all USB functions perform a complete decoding of all received bits of the PID field. If a PID field is received with incorrect values of the check bits or with undefined values of the packet type, it is assumed to be corrupted, and the remainder of the packet is ignored by the receiver.

Table 4.2 indicates the packet types, their coding and description. For simplicity, special packets are not detailed.

Table 4.2. Coding and description of USB packet types.

Packet Type	Packet Subtype	PID [3..0]	Description
Token	OUT	0001	Address and endpoint number in an output transaction
	IN	1001	Address and endpoint number in an input transaction
	SOF	0101	Start of frame marker and frame number
	SETUP	1101	Address and endpoint number in a control transaction in the setup stage
Data	DATA0	0011	Identifier for data packets with even number
	DATA1	1011	Identifier for data packets with odd number
	DATA2	0111	Identifier for data packets in high-speed isochronous input transactions with high bandwidth
	MDATA	1111	Identifier for data packets in high-speed isochronous output transactions with high bandwidth
Handshake	ACK	0010	Acknowledgement of error-free receive of data packet
	NAK	1010	Receiving device cannot accept data or transmitting device cannot send data
	STALL	1110	Endpoint is halted
	NYET	0110	No response received yet from the receiver
Special		XY00	Identifier of a special packet; XY can be 01, 10, or 11

Address Field

The address (ADDR) field specifies the address of the USB function that is the source or destination of a data packet. This field is 7 bits in length, allowing to specify up to 128 addresses. Each address defines a single function. Address 0 is reserved as the default address and may not be assigned explicitly to a function. Upon power-up and reset of a function, its

address will have the default address of 0. The host computer must set the function's address during the enumeration process.

Endpoint Field

The endpoint (ENDP) field allows more flexible addressing of functions in which several endpoints are required. This field is 4 bits in length, which allows addressing of up to 16 endpoints. Low-speed devices may only have two additional endpoints beyond the endpoint with number 0.

Data Field

The data field may contain between zero and 1024 bytes, depending on the transfer type. Data bits within each byte are sent over the bus with the least significant bit first.

Cyclic Redundancy Check Fields

These fields contain the cyclic redundancy check (CRC) codes used to verify the integrity of various fields in the token and data packets. The PID field is not included in the CRC code of a packet. The CRC codes for token and data packets ensure detection of all single-bit and double-bit errors. If the CRC code computed by the receiver differs from the code sent in a CRC field, the receiver will ignore the protected fields and, in most cases, the entire packet. The USB standard specifies the generator polynomials used for computing CRC codes. For token packets a five-bit CRC field (CRC5) is used, and for data packets a 16-bit CRC field (CRC16) is used.

End-of-Packet Field

The end-of-packet (EOP) field indicates the end of a packet by the value 0 during a two-bit period, followed by the value 1 during a one-bit period.

4.6.2. USB Packet Formats

This section presents the format of token, SOF, data, and handshake packets.

Token Packets

These packets are sent only by the host computer. The structure of a token packet is illustrated in Figure 4.15.

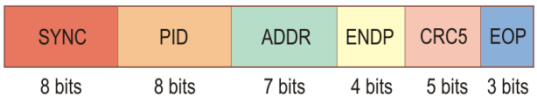


Figure 4.15. USB token packet structure.

The packet identifier field, PID, can specify a token packet with the IN, OUT, or SETUP subtype. Packets with the IN or OUT subtype inform the USB device on the direction of transfer that follows, input (reading by the host computer), or output (writing by the host computer), respectively. A packet with the SETUP subtype is used at the beginning of control transfers. For packets with the OUT or SETUP subtype, the ADDR and ENDP fields uniquely identify the endpoint that will receive the next data packet. For a packet with the IN subtype, the ADDR and ENDP fields identify the endpoint that will send a data packet. The CRC5 field contains the CRC code for the ADDR and ENDP fields.

SOF Packets

For synchronization of the entire USB system, the host computer sends a SOF (*Start Of Frame*) packet at each and every period corresponding to the beginning of a frame or micro-frame. A frame represents a time interval of 1 ms ± 0.0005 ms and is defined for the full-

speed bus (12 Mbits/s). A micro-frame represents a time interval of 125 μs ± 0.0625 μs and is defined for the high-speed bus (480 Mbits/s). A SOF packet consists of a synchronization field, a PID field, and a field of 11 bits representing the frame number, as illustrated in Figure 4.16. For the high-speed bus, the frame number will be the same for eight consecutive SOF packets, for a period of 1 ms.

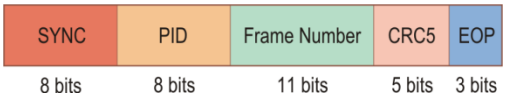


Figure 4.16. USB SOF packet structure.

All the USB functions and hubs receive the SOF packets. The reception of these packets will not initiate the generation of handshake packets by the receiver.

Data Packets

The actual information is sent over the bus in the data packets. A data packet consists of the synchronization field SYNC, a packet identifier field PID, a data field, a 16-bit CRC field, and the end-of-packet field EOP (Figure 4.17). The CRC code is computed only from the data field. The data are sent in integral number of bytes. For low-speed devices the maximum length of the data field is 8 bytes. For full-speed devices (12 Mbits/s) the maximum length of the data field is 1023 bytes, and for the high-speed devices (480 Mbits/s) the maximum length is 1024 bytes.

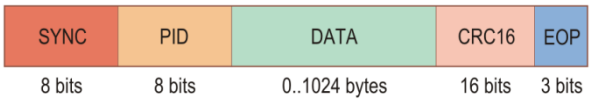


Figure 4.17. USB data packet structure.

Handshake Packets

Handshake packets only consist of the synchronization field SYNC, a packet identifier field PID, and the end-of-packet field EOP (Figure 4.18). These packets are used to report the status of a data transaction by the subtype returned in the PID field. The subtype of a handshake packet can be ACK (*Acknowledge*), NAK (*Negative Acknowledge*), STALL (*Stall*), or NYET (*No Response Yet*). These subtypes are described in Table 4.2.

Figure 4.18. USB handshake packet structure.

4.7. USB Descriptors

USB devices have a hierarchy of descriptors which describe their attributes. These descriptors are used to report the device attributes to the host computer. A descriptor represents a data structure with a format defined by the USB standards. Each descriptor begins with a byte that contains the total number of bytes of the descriptor, followed by a byte that indicates the descriptor type. In addition to the standard descriptors, USB devices may also return descriptors that are specific to a device class or a vendor.

The main types of standard descriptors are the following:

- Device descriptors;
- Configuration descriptors;
- Interface descriptors;
- Endpoint descriptors;

- String descriptors.

The descriptor hierarchy has as root at the highest level the device descriptor. At the next level, the configuration descriptors are located; there is one such descriptor for each configuration of the device. For each configuration there might be one or more interface descriptors, depending on the number of interfaces of the particular configuration. Finally, for each endpoint of each interface there is an endpoint descriptor (Figure 4.19).

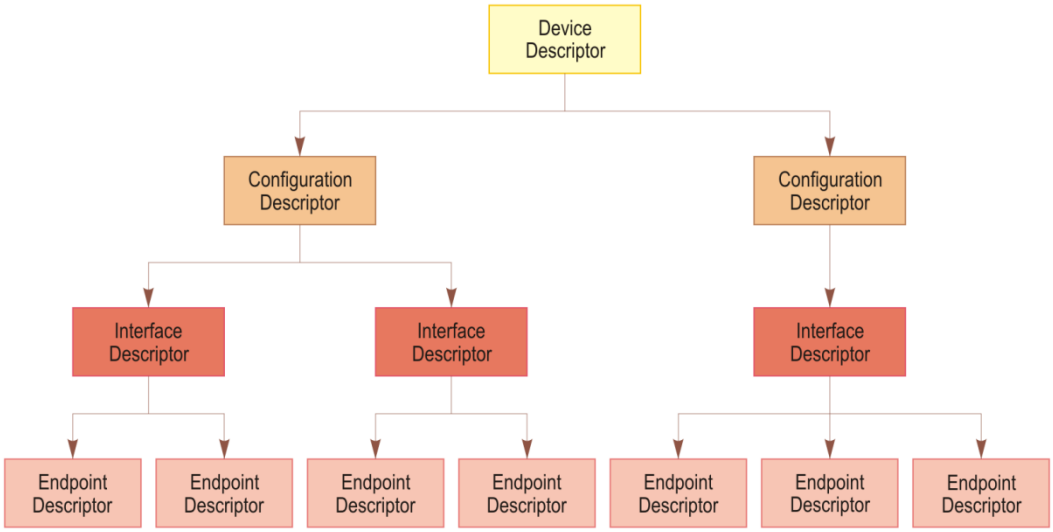


Figure 4.19. Descriptor hierarchy of a USB device.

The *device descriptor* contains general information about the USB device; this information applies to all the device's configurations. A USB device can only have one device descriptor. The device descriptor includes information such as the USB standard's revision the device complies with, the class the device belongs to, the device subclass, the vendor identifier (assigned by the USB-IF organization), the product identifier (assigned by the vendor), the number of possible configurations of the device.

A USB device can have several configurations, characterized by different attributes. A *configuration descriptor* characterizes a certain configuration of the device. This descriptor contains information such as the number of interfaces of the configuration, whether the device is powered from the bus or from its own power supply, and the maximum current drawn by the device when it is powered from the bus.

During the enumeration phase, the host computer reads the configuration descriptors and based on them enables a certain configuration of the device. Only one configuration can be enabled at a time. For example, a device might have a bus powered configuration and a self-powered configuration. If the device is connected to a computer with a mains power supply, the device driver may select the first configuration, enabling the device to be powered without connecting it to a separate power supply. If the device is connected to a portable computer, the driver may select the second configuration, which requires connecting the device to its own power supply.

An *interface descriptor* characterizes a set of endpoints within a configuration, which are grouped into an interface. For example, consider a multifunctional device consisting of a fax, a scanner, and a printer. One of the interface descriptors could characterize the fax function of the device, a second interface descriptor could characterize the scanner function, and a third interface descriptor could characterize the printer function. A device could have several interface descriptors enabled simultaneously. An interface descriptor is not directly accessible, but rather is returned as part of a configuration descriptor.

An interface may contain alternate settings, which enable to modify the endpoints associated with the interface or their characteristics after the device has been configured. By default, for an interface the alternate setting with number zero is used. Alternate settings allow

modifying an interface of the device while other interfaces remain in operation, which is more advantageous than using different configurations. If a configuration contains alternate settings for one or more of its interfaces, for each setting a separate interface descriptor and its associated endpoint descriptors must be included into the configuration descriptor.

An interface descriptor contains information such as interface number, alternate setting number for the interface, number of endpoints used by the interface, class code, and subclass code (assigned by the USB-IF organization). The number of endpoints used by the interface does not include endpoint zero.

Each endpoint used by an interface has its own descriptor. In principle, an *endpoint descriptor* contains the information required by the host computer to determine the bandwidth requirements for the endpoint. An endpoint descriptor contains information such as the number (address) of the endpoint, its direction, type of transfer used for communication with the endpoint, maximum packet size the endpoint can send or receive, and time interval for polling the endpoint by the host computer for data transfers. Endpoint descriptors are not directly accessible, being returned as part of a configuration descriptor. Endpoint zero does not have its own descriptor.

String descriptors are optional. These descriptors provide information about the USB device in a form that can be displayed directly. Reference to string descriptors is made through index values in the device, configuration, and interface descriptors. If string descriptors are not used for a USB device, all references to such descriptors must be set to zero in the device, configuration, and interface descriptors.

String descriptors have a standard structure. The first byte in each descriptor indicates the size of the descriptor in bytes, and the second byte indicates the descriptor type. Within a descriptor, the character string starts at offset 2. Each character string is encoded in the UNICODE format, as defined by the Unicode Consortium (http://www.unicode.com). The character strings are not NULL-terminated. The size (in bytes) of a string can be determined as L–2, where L is the size of the descriptor in bytes. Descriptors may contain character strings in various languages. When requesting a string descriptor, the desired language must be specified by a 16-bit language identifier (LANGID).

4.8. Enumeration Process

When a USB device is attached to the USB or is detached from the bus, the host computer executes a process referred to as *enumeration* to determine the changes occurred in the USB system's configuration. When a USB device is attached to the bus, the following operations are performed:

1. The hub to which the device is attached detects its attachment by means of the resistor used as bus termination at the device end. The hub informs the host computer on the change occurred. At this point, the USB device is powered from the bus and the port to which is attached is disabled.
2. The host computer determines the type of the change and the port at which the change occurred by querying the hub.
3. The host computer waits for a time of at least 100 ms for the supply voltage of the device to become stable, and then it issues a port enable and reset command to the port. In case of a high-speed device (480 Mbits/s), the device initiates a special electrical protocol to establish a link at this speed. If this electrical protocol is not initiated or it does not complete successfully, the communication will occur at full speed (12 Mbits/s).
4. After the reset procedure finishes, the port is enabled. The device is now in the default state and may draw a current of maximum 100 mA from the V_{BUS} line of the bus. The device will answer the transactions with the default address of zero.
5. The host computer requests the device descriptor, and the device sends this descriptor through the default pipe.

6. The host computer assigns a unique address to the device.
7. The host computer requests the configuration descriptors, and the device sends these descriptors to the host computer.
8. Based on the configuration information, the host computer assigns a certain configuration to the device. The device is now in the configured state and all the endpoints of this configuration are configured according to the characteristics specified in their descriptors. The device is ready for use and may draw from the V_{BUS} line of the bus the amount of current specified for the selected configuration.

Note

- The details of operations performed during the enumeration process may vary depending on the operating system.

When a USB device is removed from a USB port, the hub informs the host computer on the change occurred. The host computer disables the particular port and updates its information on the bus topology.

4.9. USB Requests

Each USB device must respond to USB requests issued by the host computer. The requests are sent by the host computer on the default control pipe, and the answer is sent by the device on the same pipe. The USB specifications define several standard requests that must be implemented by every USB device. In addition, there might be requests that are specific to various device classes. Also, device vendors can define their own requests.

The transmission and execution of a USB request may require two or three transfer stages. In the first stage, the host computer sends the request and its parameters in a SETUP packet using a control transfer. In the second stage, which is optional, data are transferred from the host computer to the USB device or vice versa. In the third stage, status information is transferred from the USB device to the host computer or vice versa. The direction of transfer in the status stage is opposite to the direction of transfer in the data stage. If the data stage is missing, in the status stage the direction of transfer is from the USB device to the host computer.

Each SETUP packet contains eight bytes. The structure of a SETUP packet is illustrated in Table 4.3. The field sizes are indicated in bytes. The bits within the byte that describes the characteristics of the request are denoted by b7..b0.

Table 4.3. Structure of a SETUP packet.

Offset	Field	Size	Description
0	bmRequestType	1	Characteristics of request: b7: Data transfer direction 0 = From the host to the device 1 = From the device to the host b6..b5: Request type 0 = Standard 1 = Class-specific 2 = Vendor-defined 3 = Reserved b4..b0: Request destination 0 = Device 1 = Interface 2 = Endpoint 3 = Other destination 4..31 = Reserved
1	bRequest	1	Request code
2	wValue	2	Parameter; varies according to request
4	wIndex	2	Parameter; typically, the interface or endpoint number
6	wLength	2	Number of bytes to transfer in the data stage

USB requests can be destined to devices, interfaces, or endpoints. Depending on the destination, the same request can have different effects.

4.9.1. Standard Device Requests

Standard device requests are presented synthetically in Table 4.4. The Data column indicates the data transferred in the data stage.

The *Get_Status* request directed to a device returns a status word from the device to the host computer during the data stage. The structure of the returned word is illustrated below.

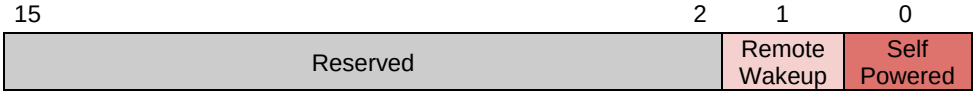


Table 4.4. Standard device requests.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0000	GET_STATUS (0x00)	0	0	2	Device status
0000 0000	CLEAR_FEATURE (0x01)	Feature selector	0	0	–
0000 0000	SET_FEATURE (0x03)	Feature selector	0	0	–
0000 0000	SET_ADDRESS (0x05)	Device address	0	0	–
1000 0000	GET_DESCRIPTOR (0x06)	Descriptor type and index	0 or LANGID	Descriptor length	Descriptor
0000 0000	SET_DESCRIPTOR (0x07)	Descriptor type and index	0 or LANGID	Descriptor length	Descriptor
1000 0000	GET_CONFIGURATION (0x08)	0	0	1	Configuration number
0000 0000	SET_CONFIGURATION (0x09)	Configuration number	0	0	–

The Self Powered bit indicates by value 1 that the device is self-powered. If this bit is 0, the device is powered from the bus. The Self Powered bit may not be changed by the *Set_Feature* or *Clear_Feature* requests. The Remote Wakeup bit indicates by value 1 that the device has the *Remote Wakeup* feature enabled. When this feature is enabled, the device can cause the host computer to pass from the inactive state (*Suspend*) into the active state. When the Remote Wakeup bit is 0, the *Remote Wakeup* feature of the device is disabled. The Remote Wakeup bit can be modified by the *Set_Feature* and *Clear_Feature* requests using the DEVICE_REMOTE_WAKEUP selector (with the value 0x01).

The *Clear_Feature* and *Set_Feature* requests directed to a device allow to disable or enable specific features of the device. The wValue field of the requests must contain the selector of the particular feature. The features that can be disabled or enabled are *Remote Wakeup*, with the DEVICE_REMOTE_WAKEUP (0x01) selector, and *Test Mode*, with the TEST_MODE (0x02) selector. The *Test Mode* feature is only implemented by high-speed USB devices and by hubs, allowing them to perform the various tests of conformity with the USB standard at the electrical interface level. The *Test Mode* feature may not be disabled by the *Clear_Feature* request.

The *Set_Address* request is used during the enumeration process to assign a unique address to the USB device. The address must be specified in the wValue field and may have the maximum value of 127. The device address will be set only after the status stage of the *Set_Address* request is completed successfully.

The *Get_Descriptor* request returns a specified descriptor if it exists. The wValue field must specify the descriptor type in the high byte and the descriptor index in the low byte. The descriptor type may specify a device descriptor (0x01), a configuration descriptor (0x02),

or a string descriptor (0x03). The descriptor index selects a specific descriptor when the device implements several descriptors of the same type. This index is only used for a configuration descriptor or string descriptor. For a device descriptor, the index must be set to 0. For string descriptors, the wIndex field must specify the language identifier (LANGID), and for other descriptors this field must be set to 0. The wLength field must contain the number of bytes to return. If the descriptor is longer than the value in the wLength field, the device returns only the specified number of bytes.

If a configuration descriptor is specified in the wValue field of the *Get_Descriptor* request, the device will return the configuration descriptor, all the interface descriptors for that configuration, and all the endpoint descriptors for all the interfaces. After the configuration descriptor, the first interface descriptor will be returned. After this descriptor, the endpoints descriptors for the first interface will follow. If there are other interfaces, their interface descriptor will be returned, followed by their endpoint descriptors.

The *Set_Descriptor* request is optional and can be used to update existing descriptors or to add new descriptors. The meaning of the request's fields is the same as for the *Get_Descriptor* request.

The *Get_Configuration* request returns the number of the current device configuration. The value is returned on a byte during the data stage. If the value returned is 0, the device is not configured.

The *Set_Configuration* request allows to enable a certain configuration of the device. The number of the desired configuration must be specified in the low byte of the wValue field. This number must match a configuration number from a configuration descriptor. The high byte of the wValue field is reserved.

4.9.2. Standard Interface Requests

Table 4.5 presents the standard interface requests.

Table 4.5. Standard interface requests.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0001	GET_STATUS (0x00)	0	Interface	2	Interface status
0000 0001	CLEAR_FEATURE (0x01)	Feature selector	Interface	0	–
0000 0001	SET_FEATURE (0x03)	Feature selector	Interface	0	–
1000 0001	GET_INTERFACE (0x0A)	0	Interface	1	Alternate setting
0000 0001	SET_INTERFACE (0x0B)	Alternate setting	Interface	0	–

For all interface requests, the wIndex field must contain in the low byte the number of the interface the request refers to.

The *Get_Status* request directed to an interface returns two bytes of 0 during the data stage. These bytes are reserved for future versions of the USB standard.

The *Clear_Feature* and *Set_Feature* requests directed to an interface allow disabling or enabling specific features of the interface. Version 2.0 of the USB standard does not allow disabling or enabling any of the interface features.

The *Get_Interface* request returns the alternate setting selected for the specified interface. The alternate setting is returned on a byte during the data stage. The principle of alternate settings has been described in Section 4.7.

The *Set_Interface* request allows to select an alternate setting for the specified interface. The wValue field must contain the alternate setting that should be selected.

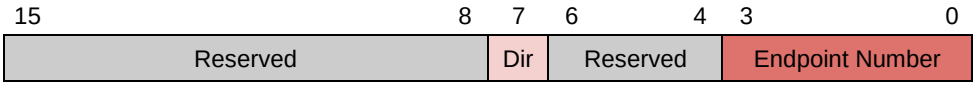
4.9.3. Standard Endpoint Requests

The standard endpoint requests are presented in Table 4.6.

Table 4.6. Standard endpoint requests.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1000 0010	GET_STATUS (0x00)	0	Endpoint	2	Endpoint status
0000 0010	CLEAR_FEATURE (0x01)	Feature selector	Endpoint	0	–
0000 0010	SET_FEATURE (0x03)	Feature selector	Endpoint	0	–
1000 0010	SYNCH_FRAME (0x0C)	0	Endpoint	2	Frame number

For endpoint requests, the wIndex field must contain the number of the endpoint the request refers to and its direction. The format of the wIndex field is illustrated below.



The Dir bit indicates by value 0 an output endpoint and by value 1 an input endpoint. For a control endpoint, the Dir bit must be set to 0. Bits 3..0 specifies the endpoint number.

The *Get_Status* request directed to an endpoint returns two status bytes during the data stage. Bit 0 of the low byte returned indicates by value 1 that the endpoint is halted (the *Halt* feature set to 1).

The *Clear_Feature* and *Set_Feature* requests directed to an endpoint allow disabling or enabling specific features of the particular endpoint. The wValue field of the requests must contain the selector of the particular feature. Version 2.0 of the USB standard allows disabling (reset to 0) and enabling (set to 1) only one endpoint feature, namely the *Halt* feature, for which the ENDPOINT_HALT (0x00) selector must be used. This feature will be reset to 0 automatically after a *Set_Configuration* or *Set_Interface* request.

The *Synch_Frame* request causes an endpoint to return the number of a synchronization frame. When an endpoint supports isochronous transfers, it may be necessary for the transfers to vary in size according to a repetitive pattern. By this request, the endpoint returns to the host computer the number of the frame in which the repetitive pattern begins.

4.10. The USB HID Class

4.10.1. Overview of the HID Class

The USB HID (*Human Interface Device*) class consists mainly of devices used by operators to control the operation of a computer system. Typical examples of devices from this class are the following:

- Keyboards and pointing devices: mouse, trackball;
- Controls on front panels: switches, buttons, sliders;
- Controls located on devices such as games or simulators: steering wheels, pedals;
- Display devices: LEDs, alphanumeric displays;
- Medical instruments: ultrasound devices;
- Devices that do not require human intervention, but which can send data in a format similar to HID-class devices: barcode readers, thermometers, measuring instruments.

Hence, the HID class consists of a large category of devices, with various characteristics. Many devices that are not provided with a human interface can also be included into this class. By using the HID-class model, it is possible to communicate in a unitary way between the host computer and various devices, which allows for a simpler development of a diversity of applications. Various operating systems and, in particular, *Windows* operating systems, provide drivers for HID-class devices, which can be used for communication with these devices. Using these drivers is advantageous, because this way the need to write drivers for communication with those devices is eliminated.

As any USB device, a HID-class device can be the source or destination of a transaction in each frame (1 ms) or micro-frame (125 μs). A transaction can contain multiple packets but is limited in size to 8 bytes for low-speed devices, 64 bytes for full-speed devices, and 1024 bytes for hi-speed devices. For hi-speed devices, it is possible to execute two or three transactions in each micro-frame. A transfer represents several transactions that create a data set with a particular structure and meaning for the device. With HID-class devices, a transfer is called *report*. The report data have a certain structure, which is specified in the report descriptors (Section 4.10.2).

There are three types of reports: input, output, and feature. An *input report* is sent by a device and contains data intended for applications ran on the host computer. Such data are, for instance, the *x* and *y* coordinates from a pointing device. An *output report* is sent by the host computer to a device and contains application data intended for controls or displays. A *feature report* contains data intended for a device or data representing the status of a device. Unlike the data in the input or output reports, data in a feature report are intended for use by the device configuration tools and not by the applications. For instance, the value of a key repeat rate can be a datum in a feature report.

Some devices may have multiple report structures. For instance, a keyboard with an integrated pointing device may independently report data referring to keys pressed and data referring to coordinates over the same endpoint. To differentiate these structures, a report identifier is used, which is a one-byte value preceding each report. For the previous example, the identifier allows the HID-class driver to distinguish key data from coordinate data.

The disadvantage of using the HID class is that typical drivers of this class support only interrupt transfers and the maximum transfer rate is limited below the USB bandwidth. The transfer rate is limited to 8,000 B/s (8 B/msec) for low-speed devices and 64,000 B/s (64 B/msec) for full-speed devices. For hi-speed devices, assuming that three transactions are executed in each micro-frame, the transfer rate is limited to approximately 23.4 MB/s (3*1024 B/micro-frame, or 3*1024*8 = 24,576 B/msec).

4.10.2. HID-Class Specific Descriptors

Besides the standard descriptors used for all USB device classes, there are specific descriptors used for the HID class. Such descriptors are the HID descriptor, the report descriptor, and the physical descriptor. Figure 4.20 illustrates the position of these descriptors in the standard USB descriptor hierarchy.

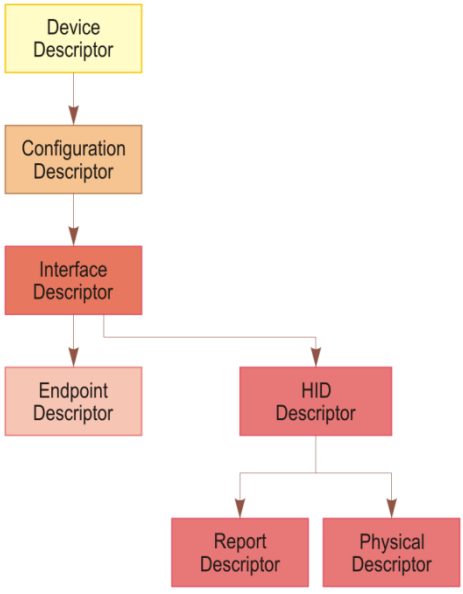


Figure 4.20. Position of HID-class specific descriptors in the standard USB descriptor hierarchy.

A HID descriptor contains the number, length, and type of subordinate HID-class specific descriptors for a device. The number of subordinate descriptors of a HID descriptor must be at least 1, since a report descriptor must always exist. The HID descriptor also contains information such as the HID class specification version the device complies with and code of the country the device is localized for.

A report descriptor contains articles that describe the size and structure of the data reported. Such a descriptor provides information about the data reported by the device for each of its functions and about the data intended for the device's functions. Examples of information are the size of data returned, whether the data are absolute or relative, and the minimum and maximum values of the individual data items. Also, a report descriptor indicates the nature of the data reported, e.g., whether they represent the *x* and *y* coordinates.

Physical descriptors are optional. These descriptors provide information about the specific part or parts of the human body used to activate the device controls. In addition, a physical descriptor may contain values that quantify the effort the user must employ to activate various controls.

4.10.3. HID-Class Specific USB Requests

HID-class devices must implement the *Get_Descriptor* and *Set_Descriptor* standard USB requests. In addition to these requests, HID-class devices may implement some requests that are specific to this class. These requests initiate transactions that allow the host computer to determine the capabilities and state of a device and to set the state of output and feature articles. The main HID-class specific requests are *Get_Report* and *Set_Report*. Implementation of the *Get_Report* request is mandatory.

Table 4.7 presents the *Get_Report* and *Set_Report* requests.

Table 4.7. The *Get_Report* and *Set_Report* requests specific to HID-class devices.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
1010 0001	GET_REPORT (0x01)	Report type and Report ID	Interface	Report length	Report
0010 0001	SET_REPORT (0x09)	Report type and Report ID	Interface	Report length	Report

The *Get_Report* request allows the host computer to receive a report from a device via the default control pipe. The wValue field must contain the report type in the high byte and the report identifier (report ID) in the low byte. The report type specifies an input report (0x01), an output report (0x02), or a feature report (0x03). If report identifiers are not used, the low byte of the wValue field must be set to 0.

The *Get_Report* request is useful at device initialization time for determining the state of its features. The request is not intended for polling the device state at regular intervals. For repeated input reports, an interrupt IN pipe should be used. Optionally, for output reports an interrupt OUT pipe may be used (Section 4.10.4).

The *Set_Report* report allows the host computer to send a report to a device for setting the state of some controls. The meaning of the request fields is the same as for the *Get_Report* request.

4.10.4. Interfacing with HID-Class Devices

A HID-class device communicates with the driver of this class either via the default control pipe or via an interrupt pipe. The device uses the default control pipe for the following operations:

- Receiving USB requests sent by the host computer and sending the answer to these requests;
- Sending data when the device is polled by the HID-class driver via the *Get_Report* request;

- Receiving data from the host computer.

The HID-class driver uses an interrupt pipe for the following operations:

- Receiving data asynchronously from a device (data that have not been requested explicitly);
- Sending data to a device with low latency.

Using an interrupt OUT pipe is optional. If a device initializes an interrupt endpoint to the output direction, then the output reports are sent by the host computer to the device via this pipe. If an interrupt endpoint with the output direction is not available, then the output reports are sent by the host computer via the control endpoint, using *Set_Report* requests.

4.11. Communication with HID-Class Devices

When a HID-class device is connected to a USB port of the computer, the operating system performs the enumeration process, as described in Section 4.8. This process does not imply using HID class-specific or device-specific drivers. During the first part of the enumeration process, the USB driver requests the device to send the device descriptor, and then retrieves the descriptor and assigns a unique address to the device. Next, the USB driver requests the device to send the configuration descriptor and retrieves this descriptor (which includes the interface descriptors and endpoint descriptors). From the interface descriptor, the USB driver determines the device class, and if this class is HID, the USB driver hands over the control to the HID-class driver. If the interface descriptor does not specify a particular class, then the USB driver selects an appropriate driver for the device based on the vendor identifier (VID) and product identifier (PID), or requests the user to specify a device-specific driver.

After the device is recognized by the operating system as a HID-class device, system functions can be employed to write a communication application with the device. These functions use the HID-class device driver, so that there is no need to write a device-specific driver. This section describes the operations required to communicate with a HID-class device for a *Windows* operating system. These operations are different for other operating systems, such as *Linux* or *MacOS*.

Before using a HID-class device, it is necessary to establish the communication with that device, which implies several steps. These steps are described next.

1. Call the `HidD_GetHidGuid()` function to obtain the *Globally Unique Identifier* (GUID) for HID-class devices. The parameter of this function is a pointer to a structure of type `_GUID`, representing a buffer into which the GUID will be returned. This function is declared in the hidsdi.h file, which is part of Microsoft's Windows Driver Kit (WDK). For static linking, the hid.lib library file should be used. For calling the `HidD_GetHidGuid()` function, add the following lines to the source file:

```
struct _GUID GUID;
HidD_GetHidGuid(&GUID);
```

2. Call the `SetupDiGetClassDevs()` function to get information about all HID-class devices attached to the system. This function returns a handle (variable of type `HANDLE`) to the information set about the device list. The first parameter of this function is a pointer to the global identifier GUID obtained in the previous step. This function is declared in the SetupAPI.h file, and for static linking the SetupAPI.lib library file should be used. For calling the `SetupDiGetClassDevs()` function, add the following lines:

```
HANDLE PnPHandle;
PnPHandle = SetupDiGetClassDevs(&GUID, NULL, NULL,
            DIGCF_PRESENT | DIGCF_DEVICEINTERFACE);
```

If the function returns the value INVALID_HANDLE_VALUE, display an error message. In this case, the communication with the device cannot be established and the operation is completed.

3. Steps 3-9 must be repeated in a loop for each HID-class device; the iteration count may be set, for instance, to 30. Call the SetupDiEnumDeviceInterfaces() function to get information about the interface of a device from the list of HID-class devices. The first parameter of this function is the handle to the information set containing the devices for which interface information is requested; this handle has been obtained in Step 2. The second parameter is optional and can be NULL. The third parameter is the pointer to the global identifier GUID. The device for which information is requested is specified by the fourth parameter representing the index (starting with 0) in the device list. The fifth and last parameter is a pointer to a variable representing a structure of type SP_DEVICE_INTERFACE_DATA that will be completed by the function with information about the device interface. Before calling the function, the cbSize member of this structure must be set to sizeof (SP_DEVICE_INTERFACE_DATA). In case of success, the function returns the value TRUE.

   Note

   - After calling the SetupDiEnumDeviceInterfaces() function, determine the code of the last error by calling the GetLastError() function. If the code of the last error is ERROR_NO_MORE_ITEMS, exit the loop with a break instruction and continue with Step 10.

4. If the function call in Step 3 has been successful, call the SetupDiGetDeviceInterfaceDetail() function to retrieve detailed information about the interface of the device selected in Step 3. The first parameter of this function is the handle to the information set, which has been obtained in Step 2. The second parameter is a pointer to the structure with information about the device interface, structure that has been completed in Step 3. The third parameter is a pointer to a variable representing a structure of type SP_DEVICE_INTERFACE_DETAIL_DATA that will be completed by the function with detailed information about the device interface. This structure contains two members, cbSize and DevicePath, the second member being a variable-length character string terminated with a byte of zero. Before calling the function, the cbSize member of this structure must be set to sizeof (SP_DEVICE_INTERFACE_DETAIL_DATA). The DevicePath member will be filled up by the called function with the access path to the device. The fourth parameter must specify the total size of the structure mentioned previously. The fifth parameter, *RequiredSize*, is a pointer to a variable of type DWORD into which the function will store the required size of the buffer that will contain the structure with the detailed information. The sixth and last parameter is optional and can be NULL.

   Note

   - The function SetupDiGetDeviceInterfaceDetail() should be called twice, as described in the function's documentation page. The first call is made to determine the required size of the buffer into which the function will store detailed information about the interface of the device; for this call, set the third parameter to NULL and the fourth parameter to zero. After the first call, memory with the appropriate size should be allocated for this buffer and the function should be called again with the parameters set normally.

   In case of success, the function returns the value TRUE. If the function returns FALSE, display an error message, free the memory allocated for the buffer to store detailed information, and continue with the next iteration from Step 3.

5. Call the `CreateFile()` function to open the communication with the device. The first parameter of this function is the device name, represented by the access path that has been obtained in Step 4 (the `DevicePath` member of the structure filled up by the `SetupDiGetDeviceInterfaceDetail()` function). The second parameter is the access mode, which can be set to `GENERIC_READ | GENERIC_WRITE`. The third parameter is the share mode of the device, which can be set to `FILE_SHARE_READ | FILE_SHARE_WRITE`. The fourth parameter is a pointer to a structure with the security attributes; this parameter is optional and can be `NULL`. The fifth parameter represents the action to take on a device that exists or does not exist; this parameter can be set to `OPEN_EXISTING`. The sixth parameter represents the device attributes and can be zero, and the seventh and last parameter is optional and can be `NULL`. If the `CreateFile()` function returns the value `INVALID_HANDLE_VALUE`, display an error message, free the memory allocated for the buffer to store detailed information, and continue with the next iteration from Step 3.

   Note

- For some devices (e.g., mouse), the access mode should be set to 0, and the share mode should be set to `FILE_SHARE_READ|FILE_SHARE_WRITE`.

6. Free the memory allocated in Step 4 for the detailed information about the device interface.
7. If the `CreateFile()` function returned a valid file handle, determine the identifier string of the HID-class device by calling the `HidD_GetProductString()` function. The parameters of this function are the handle returned by the `CreateFile()` function, the pointer to a buffer allocated by the user into which the function will place the identifier string, and the length in bytes of the allocated buffer. Add the following lines for calling the `HidD_GetProductString()` function:

   ```
   CHAR cBuffer [256];
   bRet = HidD_GetProductString(hFile, cBuffer, sizeof(cBuffer));
   ```

   where `hFile` is the handle returned by the `CreateFile()` function, and `bRet` is a Boolean variable.
8. Compare the identifier string obtained in Step 7 with the identifier string of the device the communication must be established with. Take into account that the device identifier string is represented in Unicode. If the strings match, exit the loop with a `break` instruction and continue with Step 10. If the strings do not match, continue with Step 9.
9. Close the file opened in Step 5 by calling the `CloseHandle()` function, increment the device index, and continue with Step 3 to get information about the interface of the next device.
10. Call the `SetupDiDestroyDeviceInfoList()` function to release the memory allocated for the information about the devices. At this point, the operation to establish the communication with the device is completed.

After establishing the communication with the device, the application may retrieve input reports from the device by calling the `ReadFile()` or `ReadFileEx()` function and may send output reports to the device by calling the `WriteFile()` function. For calling these functions, the handle to file returned by the `CreateFile()` function should be used.

The `ReadFile()` and `ReadFileEx()` functions retrieve input reports from the device using interrupt transfers. This means that the device must send these reports using interrupt transfers via an input endpoint. The first byte in each report represents the report identifier. It is also possible to retrieve input reports using control transfers. For this purpose, the

`HidD_GetInputReport()` function may be used, which will send a *Get_Report* request to the device.

The `WriteFile()` function sends output reports through *Set_Report* requests. If the device does not have an interrupt endpoint with the output direction, this function uses control transfers, and otherwise it uses interrupt transfers. The first byte in each report represents the report identifier. It is also possible to use the `HidD_SetOutputReport()` function to send an output report through a control transfer.

4.12. Applications

4.12.1. Answer the following questions:

a. What are the functions of USB terminators?
b. What is the difference between asynchronous transfers and isochronous transfers on the USB?
c. What information contains a device descriptor and a configuration descriptor?
d. What are the advantages and disadvantages of using the HID-class model for communication with peripherals?

4.12.2. Create a *Windows* application for displaying the identifier strings of HID-class devices attached to the system. As model for the *Windows* application, use the AppScroll-e application 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 2019* 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. Change the active solution platform 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 all the files to the project.
5. Copy to the project folder the files from the HID8.1.zip archive, available on the laboratory web page. Add to the project the SetupAPI.h and hidsdi.h header files.
6. Specify the SetupAPI.lib and hid.lib files as additional dependencies for the linker.
7. Open the AppScroll-e.cpp source file, delete the `#include "Hw.h"` directive and add `#include` directives to include the SetupAPI.h and hidsdi.h header files.
8. In the `AppScroll()` function, delete the sequences for initializing the HW library with the `HwOpen()` function and for closing the HW library with the `HwClose()` function.
9. Select *Build* → *Build Solution* and make sure that the application builds without errors.

In the AppScroll-e.cpp source file, write a function to display the identifier strings of HID-class devices attached to the system. The input parameter of the function is the handler to the application window `hWnd`; the function returns an integer value. For writing the function, follow the steps described in Section 4.11, with the following change: in Step 8, display the identifier string obtained in Step 7 and then continue with Step 9. For details on the parameters of a function, access the Windows Hardware Developer documentation by placing the cursor inside the function name and pressing the F1 key.

After writing the function, include the call to this function in the `AppScroll()` function and then verify its operation.

4.12.3. Extend Application 4.12.2 by writing a function to establish communication with the STM32L496 Discovery development board through the USB. The input parameter of the function is the handler to the application window `hWnd`. The function is similar to the function written for Application 4.12.2, except that in Step 8 it compares the identifier string obtained in Step 7 with the identifier string of the development board rather than displaying the identifier string. The board is configured as a HID-class device with the identifier string "Keil MCB2140 HID".

Notes

- In Step 5, when calling the `CreateFile()` function, the second parameter representing the access mode should be set to `GENERIC_READ | GENERIC_WRITE`, and the third parameter representing the share mode should be set to `FILE_SHARE_READ | FILE_SHARE_WRITE`.
- Change the `hFile` variable representing the handle returned by the `CreateFile()` function to a global variable, to be used by the functions that will be written for the next applications.

The function returns the value `TRUE` if the communication with the board has been established and the value `FALSE` otherwise. After writing the function, include the call to this function in the `AppScroll()` function and display a message indicating whether communication with the board has been established. Run the application without the STM32L496 Discovery development board attached to the computer. Then attach the board to the computer through two USB cables. Run the application again and verify whether communication with the development board has been established.

4.12.4. Extend Application 4.12.3 by writing a function to read and display the status of the buttons on the STM32L496 Discovery development board through the USB. The function has no input parameters and uses the handle returned by the `CreateFile()` function in Application 4.12.3 to call the `HidD_GetInputReport()` or `ReadFile()` function to retrieve a two-byte input report. The first byte of the input report is the report identifier (0x00), and the second byte contains the status of the joystick buttons on the board. The board repeatedly sends the status byte using interrupt transfers. The five joystick buttons are assigned as follows to the bits of the status byte:

Bit 0: Left button;
Bit 1: Right button;
Bit 2: Select (central) button;
Bit 3: Up button;
Bit 4: Down button.

When a button is pressed, its status bit is 1. The function displays the status byte of the buttons and returns the value 0 if the operation completed successfully or the value 1 if the `HidD_GetInputReport()` or `ReadFile()` function completed with an error.

After writing the function, include the call to this function in the `AppScroll()` function and verify its operation by connecting the STM32L496 Discovery development board to the computer.

4.12.5. Extend Application 4.12.4 by writing a function to set the state of the two user-controlled LEDs, LD1 and LD2, on the STM32L496 Discovery development board via the USB. The input parameter of the function is a byte representing the desired state of the LEDs. The function uses the handle returned by the `CreateFile()` function in Application 4.12.3 to call the `HidD_SetOutputReport()` or `WriteFile()` function in order to send a two-byte output report. The first byte of the output report should be the report identifier (0x00), and the second byte should be the desired state for the LEDs on the board. The LD1 (orange) LED is assigned to bit 0, and the LD2 (green) LED is assigned to bit 1 of the byte representing the desired state. To light up an LED, the corresponding bit should be set to 1.

The function returns the value 0 if the operation completed successfully or the value 1 if the `HidD_Set OutputReport()` or `WriteFile()` function completed with an error.

After writing the function, include the call to this function in the `AppScroll()` function, with the desired state of the LEDs defined as a constant.

4.12.6. Extend the function for displaying the identifier strings of HID-class devices, written for Application 4.12.2, to display additional information about these devices. Perform the operations described next to determine and display the additional information. Refer to the Windows Hardware Developer pages for details about the functions that should be called.

Note

- The following operations should be performed for each HID-class device detected, after displaying its identifier string.

1. Call the `HidD_GetAttributes()` function to retrieve the attributes of the device. Before calling the function, declare a variable of type `HIDD_ATTRIBUTES` and initialize its `Size` member with the size of this variable. The first parameter of the function is the handle returned by the `CreateFile()` function, and the second parameter is a pointer to the variable of type `HIDD_ATTRIBUTES`. The function will fill up the remaining members of this variable (`VendorID`, `ProductID`, and `VersionNumber`) with the vendor ID, product ID, and revision number of the device. In case of success, when the function returns the value `TRUE`, display the device's vendor ID, product ID, and revision number.
2. Call the `HidD_GetPreparsedData()` function to obtain a pointer to a function-allocated buffer containing data from the device's report descriptor. Before calling the function, declare a variable of type `PHIDP_PREPARSED_DATA`; this is a pointer to a buffer that will be written by the function with the report descriptor data. The first parameter of the function is the handle returned by the `CreateFile()` function, and the second parameter is the address of the pointer of type `PHIDP_PREPARSED_DATA`. The function will initialize the pointer to the buffer containing the descriptor data. In case of success, the function returns the value `TRUE`.
3. If the call to the `HidD_GetPreparsedData()` function has been successful, call the `HidP_GetCaps()` function to retrieve information about the capabilities of the device. Before calling the function, declare a variable of type `HIDP_CAPS`; this is a structure that will be filled up by the function with the capabilities of the device. The first parameter of the function is the pointer of type `PHIDP_PREPARSED_DATA` that has been defined in Step 2, and the second parameter is a pointer to the variable of type `HIDP_CAPS`. In case of success, when the function returns the value `HIDP_STATUS_SUCCESS` of type `NTSTATUS`, display the following information from the `HIDP_CAPS` structure: usage page (the `UsagePage` member); usage ID (the `Usage` member); length of input reports; length of output reports; length of feature reports.
4. Call the `HidD_FreePreparsedData()` function to release the memory allocated for the buffer containing the device's report descriptor data. The parameter of this function is the pointer of type `PHIDP_PREPARSED_DATA` that has been defined in Step 2.

Bibliography

[1] Allman, S., "Using the HID class eases the job of writing USB device drivers", EDN, September 19, 2002, http://m.eet.com/media/1138623/18538-243218.pdf.

[2] Axelson, J., *USB Complete. The Developer's Guide*, Fifth Edition, 2015, http://janaxelson.com/usbc.htm.

[3] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, “Universal Serial Bus Specification”, Revision 2.0, 2000, http://www.usb.org/developers/docs/usb20_docs/usb_20_0702115.zip.

[4] Crider, M., “USB 3.1 have you confused? Here’s everything you need to know about the standard”, Digital Trends, 2015, http://www.digitaltrends.com/computing/what-is-usb-3-1-when-will-it-be-released-and-what-will-it-do-for-pcs/.

[5] DataPro International Inc., “USB 3.0 Explained”, 2015, http://www.datapro.net/techinfo/usb_3_explained.html.

[6] Dong, N., “USB Type-C: One cable to connect them all”, CNET, 2015, http://www.cnet.com/news/usb-type-c-one-cable-to-connect-them-all/.

[7] Hyde, J., “Learning USB by Doing”, Intel Corporation., http://www.devasys.com/PD11x/JHWP.pdf.

[8] Microsoft Corp., MSDN Library, 2015, http://msdn.microsoft.com/library/.

[9] Microsoft Corp., WDK and WinDbg downloads, 2015, https://msdn.microsoft.com/en-us/windows/hardware/hh852365.

[10] Peacock, C., “On-The-Go Supplement – Point-to-Point Connectivity for USB”, Beyond Logic, 2005, http://retired.beyondlogic.org/usb/otghost.htm.

[11] Peacock, C., “USB in a Nutshell – Making Sense of the USB Standard”, Beyond Logic, 2010, http://www.beyondlogic.org/usbnutshell/.

[12] Rosch, W. L., *Hardware Bible*, Sixth Edition, Que Publishing, 2003.

[13] USB Implementers Forum, Inc., “A Technical Introduction to USB 2.0”, http://www.usb.org/developers/usb20/developers/whitepapers/usb_20g.pdf.

[14] USB Implementers Forum, Inc., “Device Class Definition for Human Interface Devices (HID)”, Version 1.11, 2001, http://www.usb.org/developers/hidpage/HID1_11.pdf.

[15] USB Implementers Forum, Inc., “HID Usage Tables”, Version 1.12, 2004, http://www.usb.org/developers/hidpage/Hut1_12v2.pdf.

[16] USB Implementers Forum, Inc., “Introduction to USB On-The-Go”, http://www.usb.org/developers/onthego/USB_OTG_Intro.pdf.

[17] USB Implementers Forum, Inc., “On-The-Go and Embedded Host Supplement to the USB 2.0 Specification”, Revision 2.0, 2009, http://www.usb.org/developers/onthego/USB_OTG_and_EH_2-0.pdf.

[18] USB Implementers Forum, Inc., “SuperSpeed USB”, http://www.usb.org/developers/ssusb.

[19] USB Implementers Forum, Inc., “USB 2.0 Specification Engineering Change Notice (ECN) #1: Mini-B Connector”, 2000, http://www.usb.org/developers/docs/ecn1.pdf.

[20] USB Implementers Forum, Inc., “USB Frequently Asked Questions”, http://www.usb.org/developers/usbfaq/.

[21] USB Implementers Forum, Inc., “USB On-The-Go”, http://www.usb.org/developers/onthego/.