Hochschule Konstanz
University of Applied Sciences

Automotive Information Engineering

Bachelor of Engineering

Bachelor Thesis

Analysis of the USB-Stack to create and execute test cases USB-enabled devices and MTP

Author:

Alan Koschel

August 31st, 2017

Advisors:

Prof. Dr. Werner Kleinhempel
Hochschule Konstanz
Department of Electrical Engineering and Information Technology

Albert Schaaf, M.Sc.
Marquardt Verwaltungs GmbH
TDS-SC – Software Design

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Konstanz, August 31st, 2017 Alan Koschel

Preface

Section 1 will introduce the topic of the thesis and explain why it was necessary to deal with the issue.

Section 2 will describe the objectives.

Section 3 will mention the test and evaluation environment to demonstrate what kind of tools were used to execute the tasks and reach the objective.

Section 4 deals with the concept of Windows drivers. It is explained why drivers are important and how device drivers will be allocated to devices by INF files. This is necessary and an integral part of the following sections.

Section 5 clarifies the Universal Serial Bus, which is basically one of the main and largest topics of this thesis. It begins by explaining the basic builds and is followed by the operating principles. Therefore, the Universal Serial Bus communicates by means of specific packets, which will be transmitted through various transfer types. Each device starts with transmitting various descriptors to inform the host about itself. Due to, that the host controls the bus, the device only responds to the host's transmissions. At the end of this section, the enumeration will be illustrated, that comprise the process the host learns about the device.

Section 6 gives an understanding of the Media Transfer Protocol that was running on a development board. First the main characteristics and some comparisons to similar methods will be mentioned, followed by the operating principles. These principles comprise basic operations on a device, in fact execute operations such as renaming, deleting and copying files.

Section 7 is about the test environment that was set up to set a HMI device under test that provides the USB MTP responder stack. First of all, the concept of the testing will be described and followed by the arranged test environment. In case of USB and MTP, the specific test categories will be mentioned. Lastly, the test results of the HMI device were discussed and recommendations were made to run the failed tests successfully.

Section 8 describes the outcome of this thesis and how subsequent works can expand on this thesis.

Section 9 comprises a summary that resurvey the thesis. It gives an overview about the concrete task, how the separate tasks were carried out to achieve the outcome and a summary of the outcome itself.

List of Figures

List of Tables

Contents

1 Introduction

The Universal Serial Bus (USB) is a worldwide standard for communication between peripherals. Nowadays USB interfaces are integrated in almost every device. It will be used to connect peripherals and computers. USB devices communicate between pieces of hardware, i.e., cable, plug and socket. Thus, there exists different standardized communication protocols depending on the application. In case of different communication protocols, it is necessary to verify them, that devices, no matter of country, can communicate to each other.

The verifying process is very important in order that companies can sell products with such interfaces and their designated logo, to guaranty a certain standard, which is provided all over the world. Devices have to complete various test procedures to get certified. Otherwise a company is not allowed to use logos ore designations, i.e., USB or information about data rates, i.e., *SuperSpeed*. Furthermore, successfully completed test procedures prove that a device works properly based on a professional method.

The Human-Machine-Interface (HMI) device family from the company *Marquardt Verwaltungs GmbH*, is using the USB interface for service and data exchange purposes. The service application is realized through a Virtual COM Port (VCP), based on the Communication Device Class (CDC) of USB. On the other side they want to use the Media Transfer Protocol (MTP) based on the Still Image Capture Device class for data exchange between the HMI device and a computer. Of course, the integrated circuit, which implements the USB interface on the circuit board of the HMI device has to be verified, too. The verification will be performed through an external company. In contrast, the communication protocols do not need a verification but must be examined. The identification of an USB class in an operating system does neither guaranty a proper functionality nor comply with a professional scientific method.

To accelerate the development of a project as well as to reduce the production costs, it is a significant advantage to own a test environment. Microsoft provides the possibility to verify devices on Windows operating systems. Therefor they invented the Windows Certification Program, which contains software that can be used for verification purposes. One of them is the Windows Hardware Certification Kit (HCK) we want to set up and set the HMI device under test, to examine the implementation of MTP.

Thus, it is possible to use the HCK test setup during a development process to examine a current implementation without a big effort, i.e., cooperation with an external company or similarly approaches, which delays the whole development process by far.

2 Concrete Task

Vendors must fulfill several requirements when they want to develop USB applications. Understand the USB stack, as well as their classes, is one of the first challenge. Besides from that, it is necessary to verify these applications to guarantee a proper functionality, as well as it is compatible all over the world.

Windows provides a Hardware Certification Kit to test various devices and is accessible by each vendor.

The objective of this thesis is to provide an overview about the USB stack and MTP that explains their functioning in a clear manner. Based on the new findings, a HMI device, running MTP, must be tested to verify a proper functionality. Therefor the Windows Hardware Certification Kit must be set up. The device must be set under test and the test results must be analyzed and documented.

3 Test and Evaluation Environment

To achieve the understanding of USB and MTP, several tools were used in combination. On the one hand a development board was used to implement USB functionalities and on the other hand some tools were used to analyze and understand those implementations and to illustrate the communication.

3.1 STM32F429IDISCOVERY Development Board

This is a development board (Figure 1) from STMicroelectronics, based on the STM32F429ZITx microcontroller unit that is in turn based on the ARM Cortex-M4 core[1]. It supports several input/output (IO) types, but rather within the scope of this paper it is used to implement MTP and to handle the programmable USB 2.0 receptacle. Its USB interface supports *FullSpeed* in device and host mode as well as *HighSpeed* in device and host mode, in turn simulated through a *FullSpeed* physical layer.

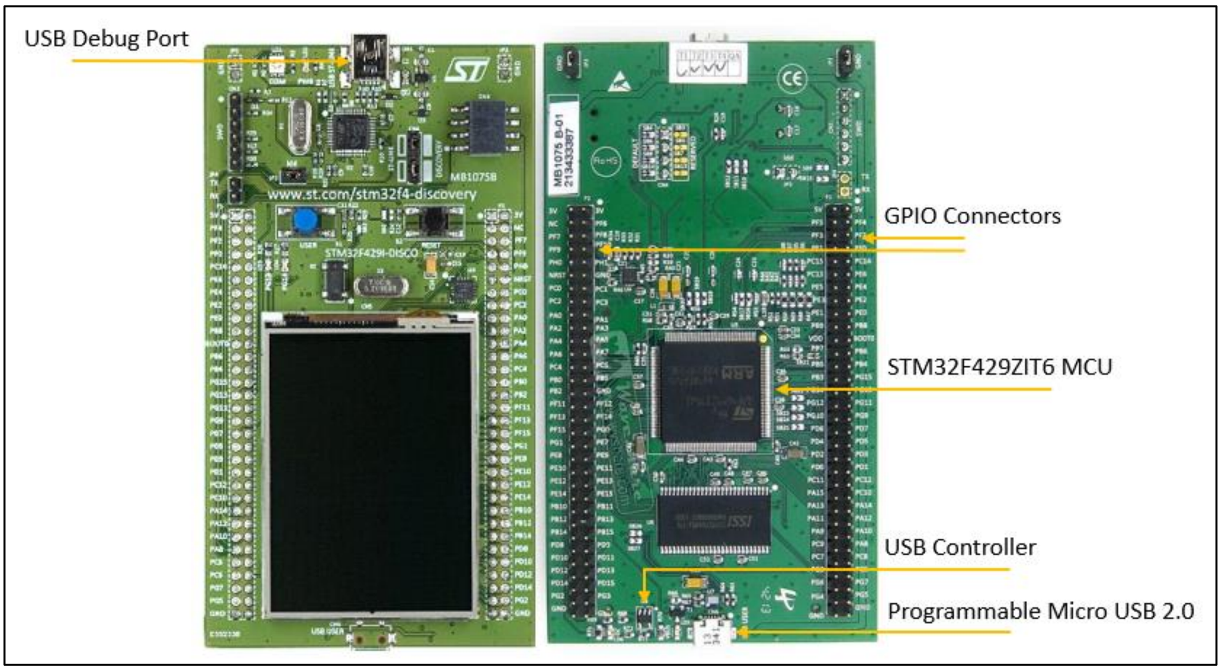


Figure 1: STM32F429IDISCOVERY Development Board

By means of the Integrated Development Environment (IDE) µVision from Keil, embedded software was implemented on the development board.

[1] http://www.st.com/content/st_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/32f429idiscovery.html

3.2 USB to Serial Cable

This cable is used to receive serial Universal Asynchronous Receiver Transmitter (UART) communication via USB. On the one hand the cable provides basically at least access to UART transmit (Tx), receive (Rx), ground (GND) and integrated circuit (IC) power-supply (VCC) connections as seen in Figure 2. For serial data transmission we will only use the Tx and Rx pin because the device is already power supplied.

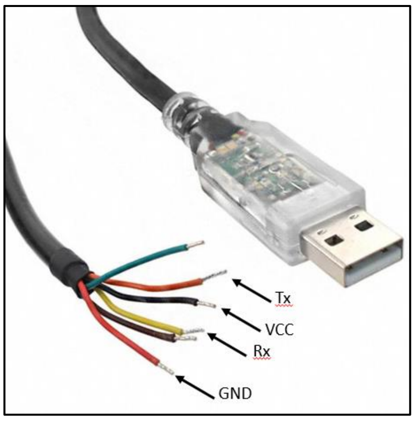


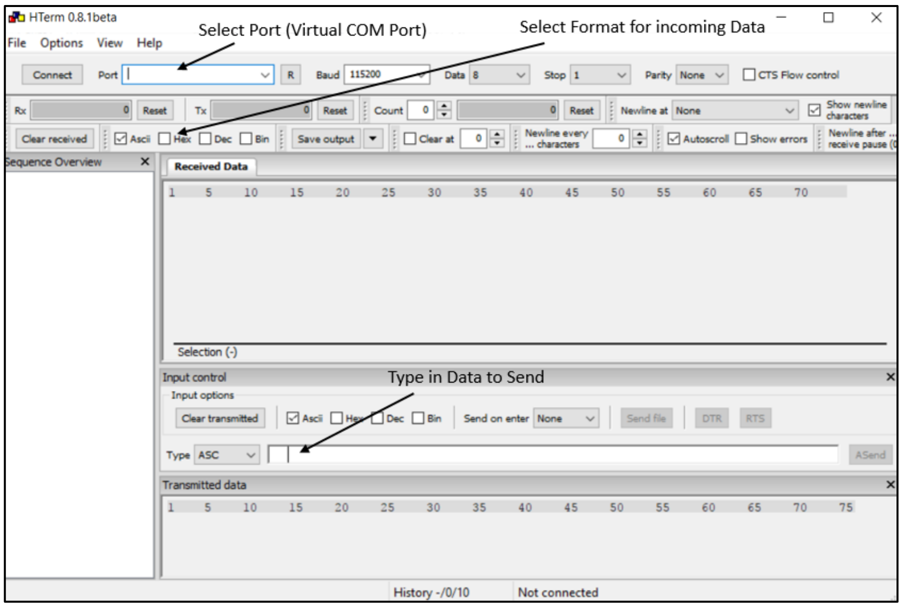
Figure 2: USB to Serial Cable

On the other hand, the USB connector contains an IC which allows to appear as a VCP. By attaching the USB connector to a computer, then a serial port will be emulated and allows to communicate through serial data with the USB interface.

Each USB to serial cable needs a specific driver to be functional, which is provided by its vendor.

3.3 HTerm

HTerm is a terminal software that provides to read data from a serial interface. Before receiving data, the COM port must be selected and connected. Additional settings are provided to select one or more data formats at the same time to show up while receiving (Figure 3).



Baud 115200 Data 8 Stop 1 Parity None ☐ CTS Flow control

Rx 0 Reset Tx 0 Reset Count 0 0 Reset Newline at None ☑ Show newline characters

Clear received ☑ Ascii ☐ Hex ☐ Dec ☐ Bin Save output ☐ Clear at 0 Newline every ... characters 0 ☑ Autoscroll ☐ Show errors Newline after ... receive pause

Input options: ☑ Ascii ☐ Hex ☐ Dec ☐ Bin Send on enter None

Type ASC

Figure 3: HTerm Terminal Software

3.4 Teledyne LeCroy Mercury T2 USB Analyzer

The hardware is based on the USB 2.0 protocol, bus powered and also operable by using any Windows PC (Figure 4).

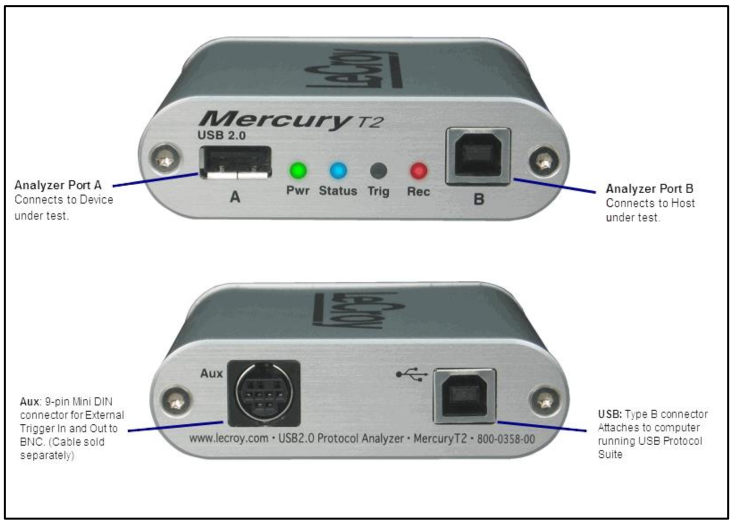
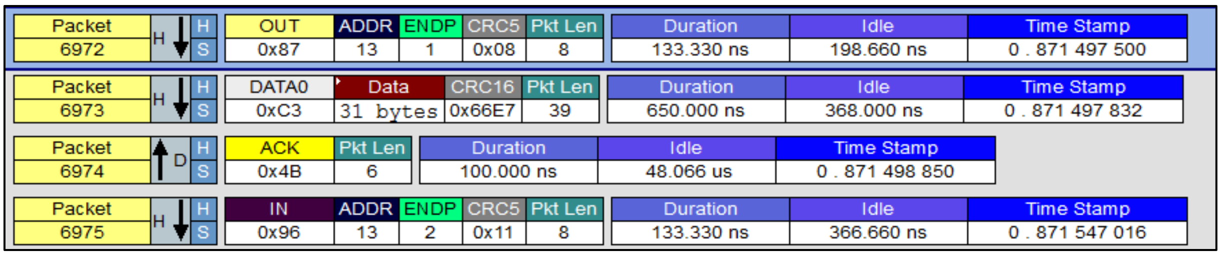

Mercury T2
USB 2.0
LeCroy
Analyzer Port A
Connects to Device under test.
A
Pwr
Status
Trig
Rec
B
Analyzer Port B
Connects to Host under test.
Aux
Aux: 9-pin Mini DIN connector for External Trigger In and Out to BNC. (Cable sold separately)
www.lecroy.com • USB2.0 Protocol Analyzer • MercuryT2 • 800-0358-00
USB: Type B connector Attaches to computer running USB Protocol Suite

Figure 4: Teledyne LeCroy Mercury T2 USB Analyzer [1]

The analyzer came in use for comprehensive and visualization purposes. It detects each frame and plot them. It decodes several USB classes, among them MTP. Logical protocol events are displayed where each frame is divided in the several transmitted packets (Figure 5).



Packet		H S	OUT	ADDR	ENDP	CRC5	Pkt Len	Duration	Idle	Time Stamp
6972	H ↓		0x87	13	1	0x08	8	133.330 ns	198.660 ns	0 . 871 497 500

Packet		H S	DATA0	Data	CRC16	Pkt Len	Duration	Idle	Time Stamp
6973	H ↓		0xC3	31 bytes	0x66E7	39	650.000 ns	368.000 ns	0 . 871 497 832

Packet		H S	ACK	Pkt Len	Duration	Idle	Time Stamp
6974	↑ D		0x4B	6	100.000 ns	48.066 us	0 . 871 498 850

Packet		H S	IN	ADDR	ENDP	CRC5	Pkt Len	Duration	Idle	Time Stamp
6975	H ↓		0x96	13	2	0x11	8	133.330 ns	366.660 ns	0 . 871 547 016

Figure 5: USB Analyzer MTP Data Packets

4 Windows Driver Concept

Understanding driver functionalities is essential to understand USB. Drivers are necessary to handle the communication between the computer and attached peripherals. Basically, Windows provides a bunch of drivers[2] to support a great variety of devices that there is no need to develop a custom driver. Windows drivers, concerning USB, are divided in USB Device Classes. These specific classes are providing support for certain USB devices, i.e., mice and keyboard, processors, sensors and especially portable devices like smartphones that are described in the Windows Portable Devices (WPD) class driver.

To develop a driver, Microsoft provides the Windows Driver Frameworks (WDF). This is for development purposes. It provides some libraries to develop drivers for devices that are fully compatible to Windows.

Development of drivers was not issue of this thesis so we will not have an in-depth discussion of this issue. WDF will be differentiated in two models, User-Mode Driver Framework (UMDF) and Kernel-Mode Driver Framework (KMDF). KMDF is focused on native Kernel calls and hides most of the operating system programming aspects. UMDF is built on KMDF and has the benefits of user mode programming compared to kernel mode programming. In conclusion a KMDF or UMDF driver is a piece of software that will be used as interface between the associated device and the computer to guaranty a proper functionality. UMDF drivers have to communicate with KMDF components to access the hardware.

In Figure 6 the USB device library architecture of STM is illustrated. The MTP responder stack (Section 6) operates in that way. An UMDF driver can be compared to the application and USB library module, referred as USB device core and USB device class. They are used to initialize any functioning the device must fulfill. Then the USB hardware abstraction layer driver module, compared to KMDF drivers will be used to pass data from the upper levels to force the hardware to operate, based on the general functioning.

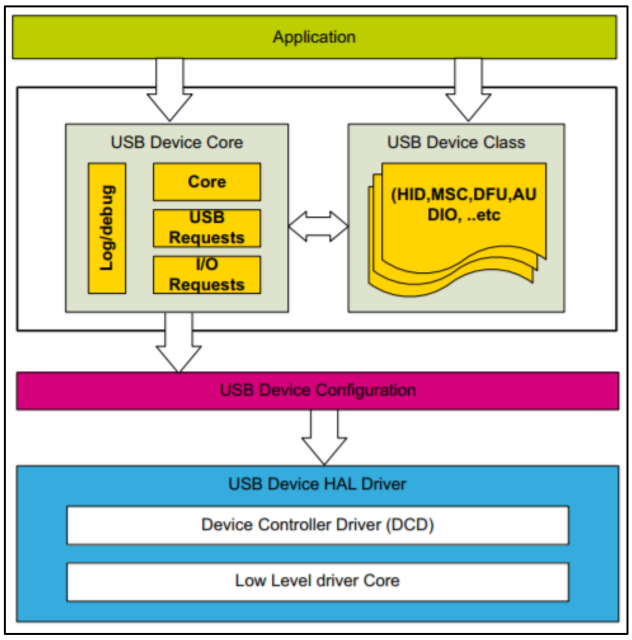
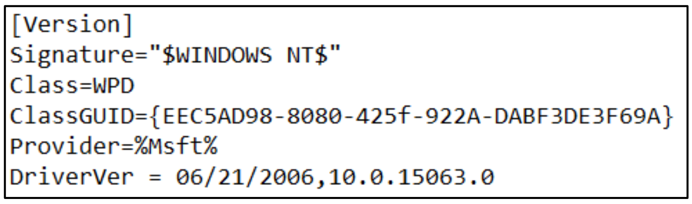


Figure 6: STM32 USB Device Library Architecture [2]

[2] https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/do-you-need-to-write-a-driver-

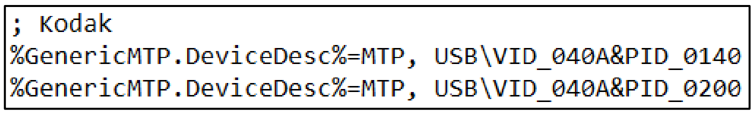
4.1 INF File

An INF file is basically a text file that contains information about a device and the components to install the specific driver during the enumeration of USB devices. Each INF file is related to an USB device class, which is described as a Global Unique Identifier (GUID) (Figure 7). The INF file contains the specific vendor ID (VID) and product ID (PID), which identify a certain device and initiate the driver install process (Figure 8). Furthermore, it contains the specific class codes concerning a protocol, e.g., MTP that helps to identify the driver, too (Figure 9).



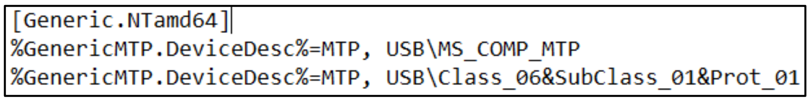
```
[Version]
Signature="$WINDOWS NT$"
Class=WPD
ClassGUID={EEC5AD98-8080-425f-922A-DABF3DE3F69A}
Provider=%Msft%
DriverVer = 06/21/2006,10.0.15063.0
```

Figure 7: USB Device Class Description of WPD in INF File



```
; Kodak
%GenericMTP.DeviceDesc%=MTP, USB\VID_040A&PID_0140
%GenericMTP.DeviceDesc%=MTP, USB\VID_040A&PID_0200
```

Figure 8: USB Device Vendor and Product ID in INF File



```
[Generic.NTamd64]
%GenericMTP.DeviceDesc%=MTP, USB\MS_COMP_MTP
%GenericMTP.DeviceDesc%=MTP, USB\Class_06&SubClass_01&Prot_01
```

Figure 9: USB Device MTP Protocol Class Codes in INF File

By developing a custom driver, it is necessary to create a custom INF file but in most cases, it is recommended to use a driver provided by Windows. Using a custom driver does not guaranty a proper functionality of a device and operating system. If a device provides more functionalities than a driver can support, it is possible to implement an extended driver. MTP driver[3] is provided by Windows and does not need any custom drivers and INF files as well.

4.1.2 INF File Recognition

Windows detects the suitable driver for an USB device through an INF file on two ways.

First, the device transmits an unique combination of VID and PID. This ID can be recognized in the specific INF file as seen in Figure 8. In case of MTP, the provided INF file does not contain a great variety of ID's and does not represent an expedient solution.

Last, the device transmits additionally the USB class code (Figure 9) to identify the used protocol. This specific class code will be recognized in the INF file and the driver will be loaded.

[3] wpdmtp.inf, wpdmtp.dll

5 Universal Serial Bus

Meanwhile USB experienced a big evolution since its release in 1996. Beginning from USB 1.0 version, providing maximum transfer rates of 1.5 Mbit/s, up to currently 20 Gbit/s in the USB 3.2 version. Apart from transfer rates, improved charging possibilities surpass the standards by USB Type-C interface, providing up to 100 W with the Power Delivery standard. The USB specifications are full of such information. The objective here is to supply a clear overview of USB 2.0 to sum up its functioning without reading the whole specification, covered in over 600 pages. It provides a transfer rate up to 480 Mbit/s and is called *HighSpeed* mode. *HighSpeed* mode is necessary to support imaging and storage features, e.g., MTP, due to USB 2.0 specification.

Furthermore, an USB stack implementation in a device is essential and defines the role it plays. Therefore, different configurations are possible and contain specific settings about descriptors, endpoints and the related communication protocol. The following outcomes were achieved through executing and analyzing the MTP responder stack [3].

But first we want to start to make clear the general structure of USB and go on with the workflow.

5.1 USB Topology

USB is arranged kind of a network comprising at least and maximum one host to which peripherals, also called devices, can be connected. The host will be generally represented by a computer, that always come into force as host and is not negotiable. A bunch of different devices, with different functionalities, e.g., mice, keyboards, printers were used by connecting through USB interface with a host. Moreover, a hub can be placed between the host and one or more devices. Hubs that are embedded in the host are also called root hub. In Figure 10 is visualized how devices are connected to a host and how host ports can be expanded.

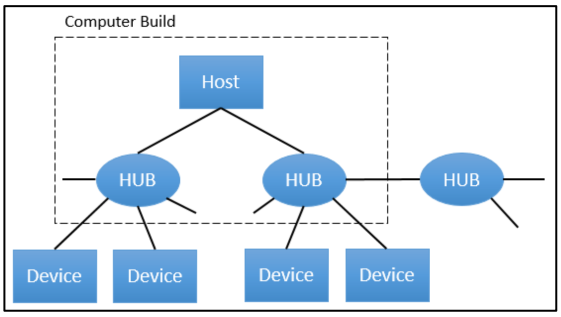


Figure 10: Host and Device Arrangement (USB)

Hubs are used to enlarge the limit of connected devices. By connecting a hub to a port on the host side, the port will be converted into multiple ports. All in all, the connection is organized through a star topology and each device is attached by one wire, called the bus, to its host.

In the end the bus is controlled by the host and triggers the whole communication whereby the connected device only response.

5.2 USB Plug and Role Allocation

We will limit this part and only mention USB type-A and micro-B plugs to clarify how devices get recognized by its host.

Type-A plugs are the most commonly used plugs and fit into the USB receptacles in each computer that takes the role of the host or any other computer-like device. It has four pins, D- and D+ for data transmission. The third and fourth pin, VCC and GND, provide bus power.

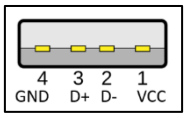


Figure 11: USB Type-A Plug Pins

Micro-B plugs are used to connect devices with a host. Common smartphones have a micro-B receptacle to which the plug can be attached. Then, the smartphone acts as device. The plug has five pins. Equal to type-A plug, the D+ and D- pins provide data transmission and power supply is provided through VCC and GND. Additionally it has an ID pin which defines the role of the attached device either as device or host (Figure 12). Actually, the ID pin in this cable is always connected to GND, so the role as device is predefined.

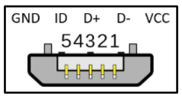


Figure 12: USB Micro-B Plug Pins

In conclusion, then, it is clear that peripherals, attached to USB micro-B connectors are always recognized as device. On the other side, that means that it cannot operate as host with this type of cable. It is not possible to connect two host devices with each other. Host and device detection will be realized by electrical properties. Therefor the host and the device are pulling the data lines with different resistance. Devices with an USB micro-B receptacle are predefined to act as a device. In case of an USB micro-AB receptacle, the device is able to act as host as well as a device. But in the end, it depends on the connector. If both devices provide an USB micro-AB receptacle, the Host Negotiation Protocol[4] of the USB On-The-Go specification arrange the roles, which additionally depends on the application of both devices.

5.3 USB Packet Fields and Types

Transmitted USB messages, which are subdivided in packets, always begin with the least significant bit. A message consists of various fields and is defined as a specific type.

5.3.1 Packet Fields

The following field types can occur in a message but the appearance of each field type in one message is not mandatory (Table 1).

Table 1: USB Packet Fields

Packet Field	Description
Sync	The Sync field is 32-Bit long for High Speed and synchronize the clock of the receiver with the clock of the transmitter.
PID	This is the Packet ID and indicates the type of the packet which is being sent.
ADDR	The address specified which device is intend to receive the packet. It consists of 7 Bit which leads to that up to 127 devices are addressable. Devices that are not assigned are bound to respond on packets addressed to zero.
ENDP	This field addresses the device related endpoint.
CRC	The Cyclic Redundancy Check is placed in token packets as well as in data packets.
EOP	This packet announces the end of a packet.

[4] http://www.usb.org/developers/onthego/USB_OTG_and_EH_2-0.pdf

5.3.2 Packet Types

Transmitted messages are distinguished in different packet types (Table 2).

Table 2: USB Packet Types

Packet Type	**Description**
Token	This is always the first packet in a transaction determines the address, endpoint and purpose of the following transaction. The purpose is either indicated as IN, OUT or SETUP. The IN token informs the device that the host wants to read information. The OUT token informs the device that the host wants to send data to the device. The SETUP token introduces a control transfer. The message is structured as follows: SYNC | PID | ADDR | ENDP | CRC5 | EOP
Data	The data packet is going to be send after an IN or OUT packet. Its length depends on the data size and is limited by 1024 bytes in High Speed mode. If the admissible data size is bigger than the limit, the data will be separated and send over various messages. Then each data packet comprises the full data size except the last one. The message is structured as follows: SYNC | PID | DATA | CRC16 | EOP
Handshake	A handshake packet signals the status of an endpoint, i.e., transaction passed or failed. The handshake is subdivided into three types. An ACK is an acknowledgement and signals a successful received message as well as the endpoint is ready to receive new data. A NAK signals that the endpoint is currently not able to send or receive new data, due to it, it is still processing data or does not processed the data before yet. A STALL signals that the endpoint remains in a state where it is halted. The message is structured as follows: SYNC | PID | EOP

5.4 Endpoints

An endpoint (EP) is comparable to a buffer. It buffers data that comes from the host or will consumed by the host. The endpoint operates between host and device, but is logically placed on the device. If the host sends data to the bus, it reaches not directly the device but its EP OUT buffer. Then the software of the device will read the received data on EP OUT buffer. In case that the device wants to response to the host on the received data, it cannot write data to the bus immediately, because the bus is host controlled. Furthermore, it cannot write data to the same buffer, in case the host sends again data to the bus that will end up in the same buffer. Therefore, the device needs a second buffer, called EP IN buffer. This buffer is responsible to store data that is going to be consumed by the host.

By means of Figure 13 we want to illustrate how a communication process could look like. First, the host needs to determine to send or receive data. In case of sending data, the host is going to send a token packet, including an OUT token. Based on the address, the device recognize that the incoming data is for itself and buffers it in the EP OUT buffer. Then an ACK will be send to notify the host that the message was successfully received. The Interrupt Handler of the device is going to trigger an interrupt to notify the software application that data had arrived.

After the data was processing, the device can send a response if required. First the response data will be stored in the EP IN buffer. The data will remain in the buffer until the host is going to send a token packet, including an IN token. Due to the address, related to the device, it recognizes the IN token and reply with the data placed in the EP IN buffer. According to this procedure the host will be reply with an ACK.

Referring to receive and buffer data and also triggering an interrupt, this takes place in hardware, usually processed by an USB controller.

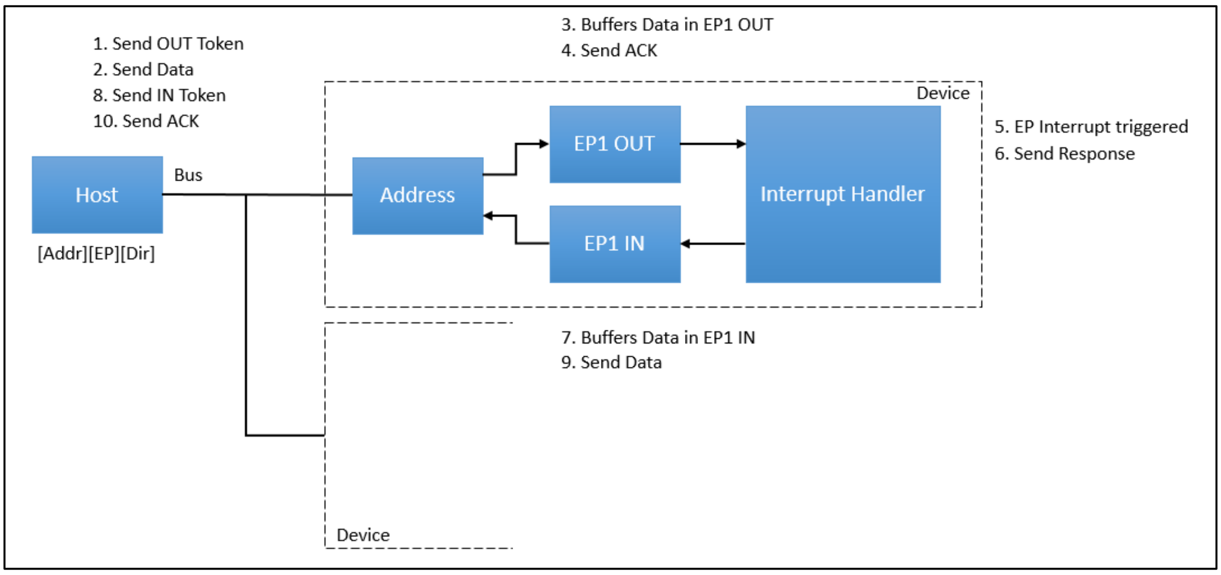


Figure 13: USB Endpoint Communication Model, inspired by [4]

No matter how many endpoints a device provides, it must provide at least one control endpoint, which is referred as EP0. The control endpoint is reserved for setup configurations, called control transfer (Section 5.5.1). By connecting a device to a host, the host requests device information, configure the device or process control tasks. The control endpoint will be used during the enumeration process (Section 5.9) and while the device operates on the bus.

To sum up, an endpoint usually contains two buffers, i.e., one for read and one for write operations related to the bus. In contrast, the unique EP0 must be supported by every device as soon as a connection is established without any configuration and is independent from the device class as well as from other endpoints. EP0 is bidirectional, which means that it communicates through one buffer. The whole transaction stage is usually handled by hardware. Interrupt endpoints can contain either an IN buffer or an OUT buffer or both.

5.5 USB Transfer Types

USB provides four different transfer types, also referred as endpoint types. Which transfer type is going to be used depends on the functionalities of the device. For example, a bulk transfer is used to transmit insensitive data, e.g., media files on storage devices or data for printers. There are four different transfer types, but with regard to MTP we will mention three of them, the control, bulk and interrupt transfer. Otherwise the isochronous transfer guaranty a specific bandwidth and occurs continuously, specifically for audio or video streams.

The following transfer types were logged and analyzed by means of the STM32F429IDISCOVERY development board (Section 3.1) and its MTP responder stack (Section 6.5) that we put into operation [3].

5.5.1 Control Transfer

The control transfer is used for command and status operations on the default EP0, each device must provide. To enumerate a device (Section 5.9) the control transfer is used, too. A control transfer is divided in three stages.

Setup Stage

In the setup stage, three packets will be transmitted by the host. First, a packet with a SETUP token where address and endpoint number will be sent to the device. Next, a data packet will be sent containing a PID type of DATA0 and contains a setup packet that defines the type of request. The third and last packet is usually an ACK as handshake. In case of any failures, the setup packet will be ignored because it cannot be replied with a NAK or STALL. In the following Figure 14 the setup stage is illustrated.

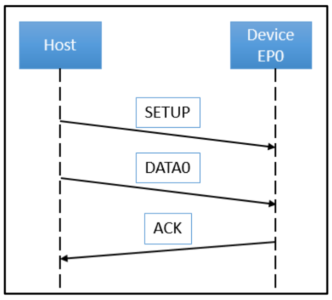


Figure 14: Communication in Setup Stage

Data Stage

This stage describes IN and OUT transfers. Depending on the direction of the data transfer, there are two scenarios. As illustrated in Figure 15, the host sends an IN token which signals a read command. Then the expected data in the IN EP buffer will be send to the host. A NAK packet indicates that the endpoint has no data and a STALL packet indicates an error. In the end the host will send an ACK. In case of an OUT token, the host wants to transmit data to the device. Thereupon the host will send the data in expectation of a handshake. The device replies with an ACK, NAK or STALL.

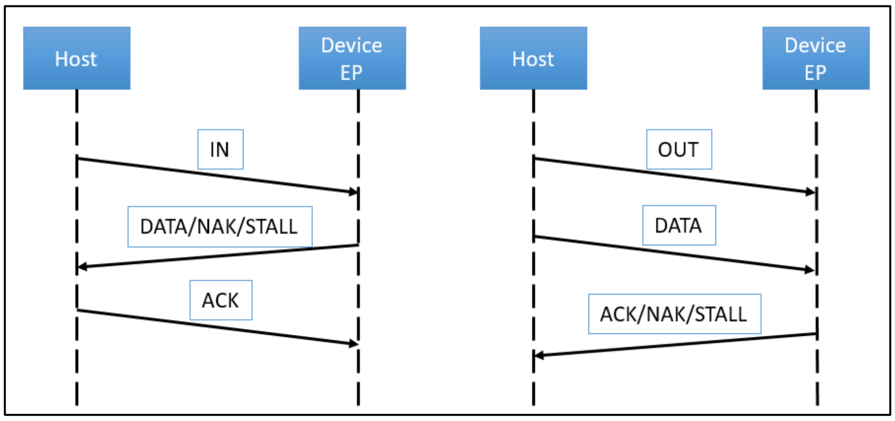


Figure 15: IN and OUT Transfer in Data Stage

Status Stage

Some packets must be confirmed. An ACK packet from an endpoint is sometimes not enough to confirm a received packet. Therefore, it is important to send a zero-length-packet (ZLP). For example, when the host sends a SET_ADDRESS request (Section 5.7), the device must confirm the received request by sending back a ZLP. This packet contains no data with a length of zero. Additionally, a ZLP can be used to indicate the end of the transfer on other endpoints. For example, when a packet will be send with the maximum of 1024 Bytes, it is necessary to send a ZLP to indicate that the last packet was the last packet containing the requested data. As seen in Figure 16, the device confirms with a ZLP after the host sends an IN token to the specific endpoint, followed by an ACK of the host. If the endpoint has no data to transfer, it sends a NAK packet, or in case of an error, it sends a STALL packet. The host confirms by sending first an OUT token to indicate the transfer of data. Then the host transmits a ZLP, followed by a handshake from the endpoint.

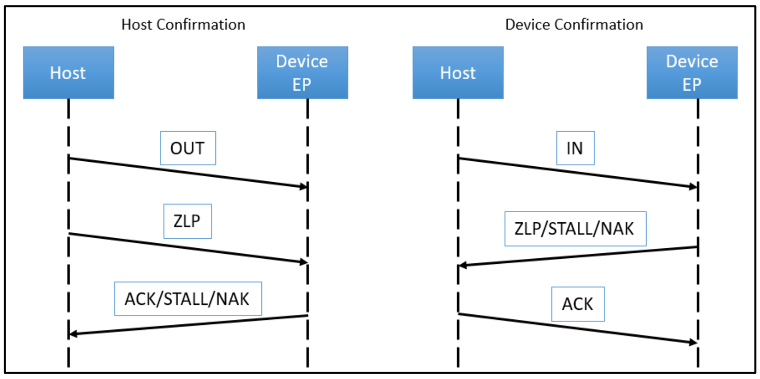


Figure 16: Host and Device Confirmation by Zero-Length-Packet

5.5.2 Bulk Transfer

Bulk transfer is used by bulk endpoints which are subdivided in IN and OUT endpoints. This transfer type is used to transmit a large amount of data, e.g., media files but in general insensitive data, because there is no bandwidth reserved on the bus. In case of multiple transactions, this type of data will be transmitted when there is bandwidth left. In conclusion, there is no guaranty of latency for bulk transfers. When there is no traffic on the bus, the transfer is fast. The bulk has a low priority on the bus. The bulk transfer operates on IN and OUT endpoints as seen in Figure 17. To initiate a bulk transfer the host first sends an OUT token in case of sending data to the device. Then the OUT token is followed by a DATA packet and will be committed by either an ACK, NAK to signal that the endpoint is not empty or a STALL in case of an error. When the host wants to read data, an IN token will be sent in order to receive data from the associated endpoint. The device can reply with the data, a STALL or a NAK to signal that there is no data to send. Finally, the host replies with an ACK.

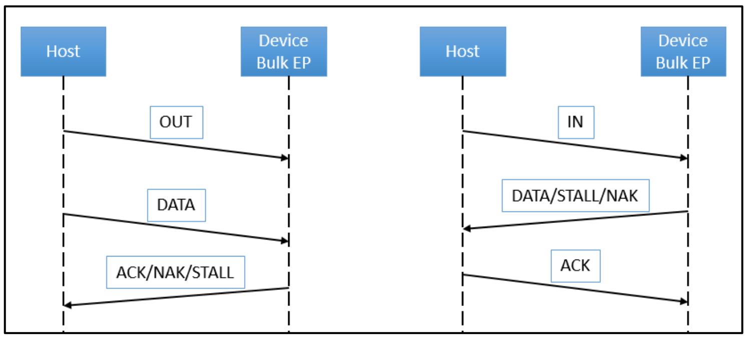


Figure 17: Bulk Transfer IN and OUT

5.5.3 Interrupt Transfer

Therefor a specific bandwidth will be guaranteed. An interrupt endpoint can consist of either an IN or OUT endpoint or both. On the development board STM32F429IDISCOVERY (Section 3.1) only an IN interrupt endpoint is implemented. Due to the bus, which is host controlled, the device must wait until the host queries it, that it can report its issues. As mentioned in Figure 18, the host sends an IN token to the IN interrupt endpoint to query the device for data. In case of issues, the device replies with data. Otherwise it replies with a NAK, because there is no data to send, or a STALL when the endpoint is halted.

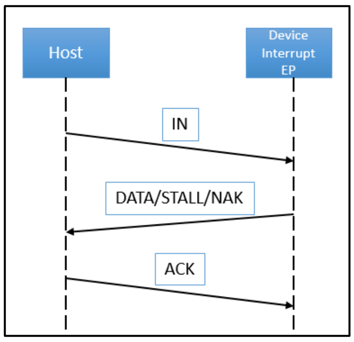


Figure 18: Interrupt Transfer IN

5.6 Descriptors

Each device is described through a hierarchical structure of descriptors as seen in Figure 19. The host learns through descriptors what kind of device it is, i.e., about the vendor, number of configurations and endpoints as well as their types. To request descriptors the host uses control transfers to transmit various standard control requests, which are described in Section 5.7.

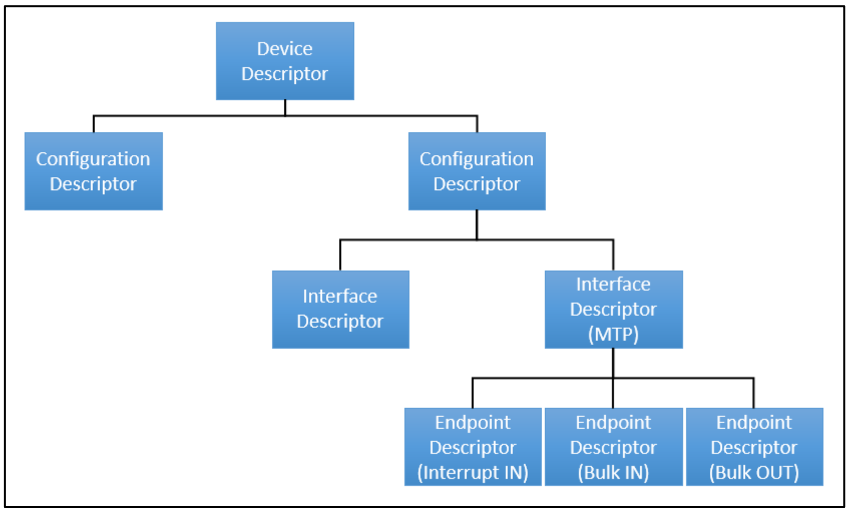

Device Descriptor
Configuration Descriptor
Configuration Descriptor
Interface Descriptor
Interface Descriptor (MTP)
Endpoint Descriptor (Interrupt IN)
Endpoint Descriptor (Bulk IN)
Endpoint Descriptor (Bulk OUT)

Figure 19: Hierarchical Structure of Descriptors

5.6.1 Device Descriptor

As seen in Figure 19 the device descriptor is on top of the hierarchical structure and will be requested first. The device descriptor releases first information about itself. This includes details like PID and VID, the number of configurations the device provides and the max packet size which can be transmitted.

5.6.2 Configuration Descriptor

The second and last descriptor that is going to be requested after the device descriptor is the configuration descriptor (Figure 19). It contains at once all the data about the followed descriptors, the interface and endpoint descriptor. It releases information about the maximum current and power supply, also the number of interfaces. A device can have more than one configuration at once, though only one can be used at a time. In case of various configurations, they will differ by current or power supply settings but it is not common to provide more than one configuration descriptors.

5.6.3 Interface Descriptor

An interface descriptor defines the function the device fulfills. Primarily it contains information about the various class codes, as described in Section 4.1, which define the usage of MTP. One configuration can provide various interfaces (Figure 19). For example, a printer provides one interface for each function, e.g. scanning, printing and faxing.

5.6.4 Endpoint Descriptor

Each endpoint is associated with an interface. The MTP responder stack [3] we put into operation on the STM32F429IDISCOVERY (Section 3.1) contains three endpoints to its interface. As illustrated in Figure 19, an IN interrupt endpoint for issues and bulk endpoints, consisting of IN and OUT buffer. An endpoint descriptor delivers information to the host about its address, type of endpoint and the max packet size of transmission. Interrupt endpoints provide additionally information about the exact interval the endpoint can be used to initiate transfers. This means that the host will send IN tokens periodically to the interrupt endpoint of the device, to read out its data that the device put into the buffer to signal any complications. Commonly interrupts will be triggered by devices itself, but in this case the host has to query a device for interrupts because the bus is host controlled.

For detail information about how descriptors can be implemented in software, the MTP responder stack [3] of the STM32F4IDISCOVERY is appended to this paper, specifically in the files *usbd_desc.c* and *usbd_mtp.c*.

5.7 Requests

Requests will be used in setup packets to request certain data. This is the method the host learns about the device through control transfer (Section 5.5.1). Each request must be replied by the device in a certain amount of time, otherwise the transfer will be aborted and the connection will fail. For example, when the host requests the device descriptor, the device must reply within 500 milliseconds to maintain the connection. In the following Table 3 the format of a request packet, also setup data, is illustrated.

Table 3: Format of Request Data [5]

Offset	**Field**	**Size [Byte]**	**Description**
0	bmRequestType	1	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6-5: Type 0 = Standard 1 = Class 2= Vendor 3 = Reserved D4-0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4-31 = Reserved
1	bRequest	1	Section 5.7.2
2	wValue	2	Varies according to request (Section 5.7.3)

Offset	Field	Size [Byte]	Description
4	wIndex	2	Varies according to request (Section 5.7.4)
6	wLength	2	Number of bytes to transfer if there is a Data stage

5.7.1 bmRequestType

This field describes the type of request. It identifies the direction of data transfer. Usually the host transmits standard requests (Table 4). In the first five bits it is defined who is the recipient.

5.7.2 bRequest

This field defines the specific request (Table 4).

5.7.3 wValue

For example, when the host wants to set the address of the device during the enumeration (Section 5.9), the address will be passed in the *wValue* field.

5.7.4 wIndex

It is used to assign an interface or endpoint. In Table 4 is seen how the data is going to be passed on.

5.7.5 wLength

This field describes the length of data containing the data packet. If the host transmits an OUT packet, *wLength* specify the exact amount of data. But in case of an IN packet, the device can reply either with exact the same length as defined in *wLength* or less than *wLength*. If *wLength* is equal to zero, there is no data stage followed.

5.7.6 Standard Device Requests

They are used to request the various requests to learn about the device and choose configurations. Furthermore, they request the current status, i.e., it is self-powered or bus-powered. The standard device requests as seen in Table 4 are used during the enumeration (Section 5.9).

Table 4: Standard Device Requests [5]

bmRequest Type	bRequest	wValue	wIndex	wLength [Byte]	Data
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status

bmRequest Type	bRequest	wValue	wIndex	wLength [Byte]	Data
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero	Descriptor Length	Descriptor
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None

5.8 Vendor and Product ID

The VID and PID inform the host about the device' vendor. A company can use the same VID for all the USB products. But each product must use a different PID. Different PID's are necessary to differ devices while connected to the same host. For example, when an USB memory stick is connected to a host and the same stick, with the same VID and PID, is going to be attached to the same host, the host cannot distinguish them and the second one would be unusable. In case of using the same combination of VID and PID, the USB class should be the same to avoid driver issues, referring to Section 4.1. VID's and PID's can be requested at the USB Implementers Forum[5].

5.9 Enumeration

When a device will be attached to a host, the host needs to gather information about the device in order to subsequently load the appropriate driver. This process is called enumeration. During the enumeration, the host communicates through control transfer by transmitting setup packets (Section 5.5.1) to learn about the attached device. With the MTP (Section 6) responder stack on the STM32F429IDISCOVERY (Section 3.1) we want to clarify the enumeration. A complete and in-depth sequence diagram about the enumeration of the STM32F429IDISCOVERY will be appended to this thesis. The device does not know anything exactly about the enumeration. It is obligated to reply the requests, send by the host, that the host can successfully enumerate the device. By attaching a device into an USB port, the port provides power supply to the device and reports the host that a device was attached to it. When the host knows where the device is attached and at which speed it operates, the enumeration starts as shown in Figure 20. The host recognize the attached device by finding out, which line has a higher voltage when idling, performed by the port.

[5] www.usb.org/developer/vendor/

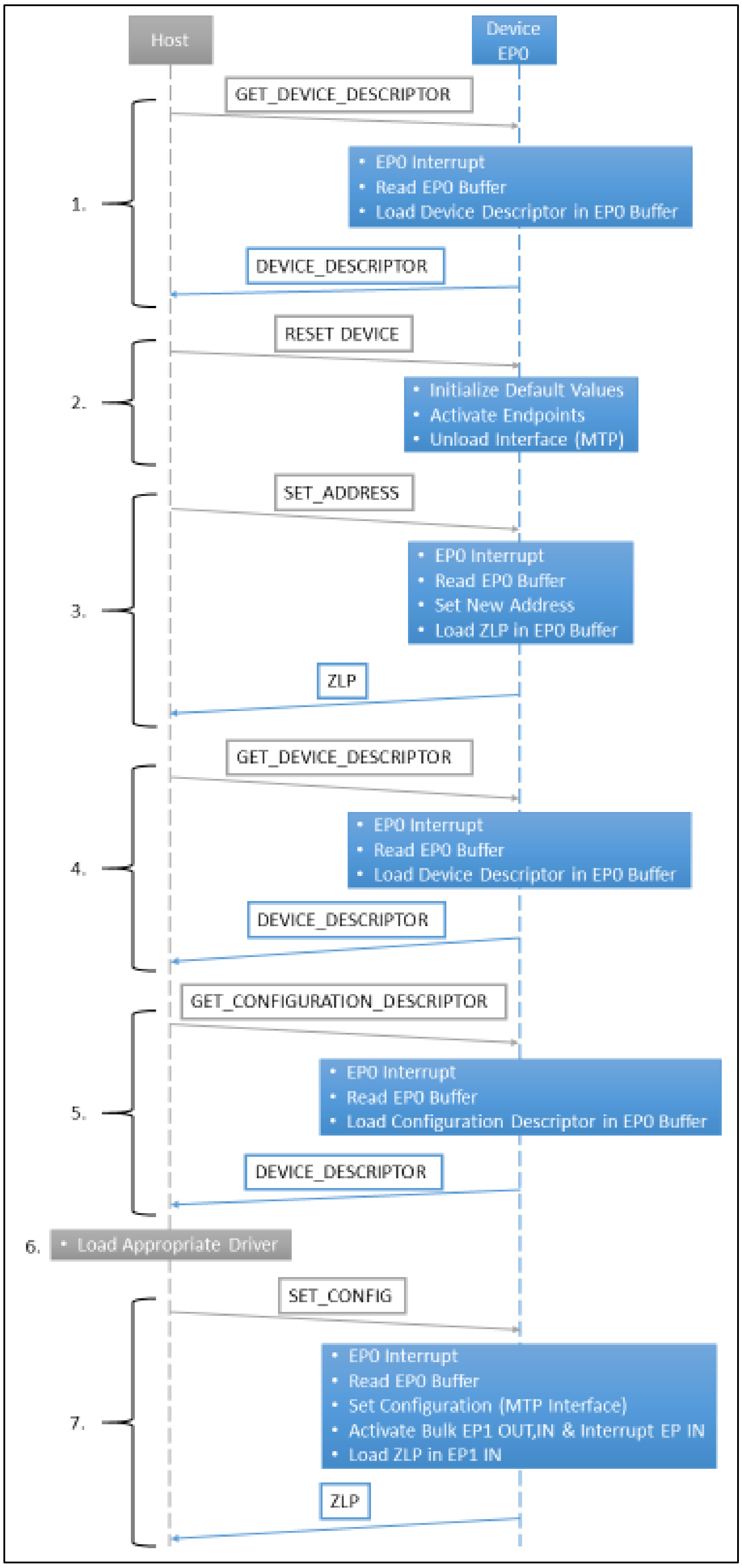


Figure 20: Enumeration Process with Focus on the STM32F429IDISCOVERY

First, the host is going to send a *GET_DEVICE_DESCRIPTOR* request to EP0, in order to figure out the maximum packet size of the default EP0. Therefore, EP0 initiate an interrupt to signal that a setup packet arrived. The EP0 buffer will be read out and the device descriptor (Section 5.6.1) will be written into the EP0 buffer. In case of several simultaneously attached devices, only one device will reply to address *00h* because the host can enumerate only one device at a time.

Second, after the host received the device descriptor, the host prompts the port to reset the device to ensure that it is in a known state when proceeding. Although it is not required by the USB 2.0 specification.

Third, the host will assign an unique address to the device by sending a *SET_ADDRESS* request. Until the request was processed, the communication takes place through the default address. EP0 generates an interrupt. Then the buffer will be read out and regarding to this request, the new address will be set. Subsequently a zero-length-packet will be loaded into the EP0 buffer to confirm the transaction. The new address will be used until the device is going to be attached or rebooted.

Fourth, by means of sending a *GET_DEVICE_DESCRIPTOR* request again, the host starts to learn about the device properties, i.e., basic information and number of possible configurations the device can have as mentioned in Section 5.6. The procedure on device side is the same as the first *GET_DEVICE_DESCRIPTOR* request.

Fifth, the host sends a *GET_CONFIGURATION_DESCRIPTOR* request to select a suitable driver for the device. On the device side an EP0 interrupt will be triggered, then the EP0 buffer will be read out. According to the request, the configuration descriptor (Section 5.6.2) will be loaded into the EP0 buffer. The configuration descriptor contains also the interface (Section 5.6.3) and endpoint (Section 5.6.4) descriptors.

Sixth, the host is looking for a suitable driver on the basis of the descriptors. If the device is already known by the host, the host will find its information in the Microsoft Windows Registry and load the assigned driver. Otherwise an INF file will assign a driver as mentioned in Section 4.1.

Seventh, the host sends a *SET_CONFIG* request to select a configuration, e.g., MTP or scanning. Therefore the EP0 buffer initiate an interrupt and then the buffer will be read out. According to the request, the appropriate configuration, in our case MTP, will be initialized. The bulk and interrupt endpoints, described in Section 5.5, will be activated and the device can fulfill its function. In conclusion, a zero-length-packet will be loaded into the IN EP1 buffer.

6 Media Transfer Protocol

MTP was developed as an extension of the Picture Transfer Protocol (PTP) and makes it possible to exchange a bunch of different file types between different devices. MTP acts as an interface and is associated with the storage of a device. Commonly smartphones and digital cameras are the main target where MTP operates as software deployment. The main objective is to release only insensitive data, e.g., videos or music files to the host without exposing the physical file system.

This is part of the thesis because *Marquardt GmbH* wants to integrate MTP into their HMI device family. The objective is to demonstrate the basic functioning of MTP by running the associated MTP responder stack [3] on the STM32F429IDISCOVERY development board (Section 3.1). In case of USB it is not possible to simply debug the running software, because the host expects responds in a certain period of time which takes place in a range of milliseconds. To gather some more detailed information during the execution of the whole functioning, some outputs where placed between the code lines. The outputs were made through UART over an USB to serial cable, which was used to connect the development board with the computer. Therefor see Section 3.

6.1 Comparison to PTP

MTP is based on PTP and extend it in a way that is capable to operate with much more different data types. PTP was designed to copy photos from a digital camera and store them for example on a computer. It is only able to provide photographic images such as JPEG files. Moreover, MTP provides a bunch of different file types, e.g., different music, video or text file types and much more. MTP, as well as PTP, is standardized by the USB Implementers Forum as Still Image Capture Device class for USB.

6.2 Comparison to Mass Storage Device Class

The Mass Storage Device Class (MSC) takes the lead at compatibility. Whereas MTP has a lack of supported devices, MSC is nearly supported everywhere. Devices like DVD players or car stereos do not support MTP and are only capable to read files from a MSC device, e.g., USB memory sticks. Some operating systems, e.g., Windows 7 and later versions support MTP natively. Other operating systems, e.g., Linux or MacOS, can be updated to support MTP.

MSC devices represent the storage itself and expose it to the host. That means that data will be read or written directly on the storage system. Before MTP was integrated into mobile devices, their storage was handled as USB MSC. Therefor commonly a mobile device had two different storages. One storage for sensitive data and applications and the second storage for media data, accessible by the host. From the beginning, a vendor had to define how much storage the device will provide for sensitive data and on the other side for media data. Those limitations can lead to storage problems and since MTP, the storage will be represented as one storage for all data. MTP does not represent the storage itself but provides an interface to it. In contrast to MSC, the user of a portable device can access to it while it is connected to a host.

Besides the MSC device, a MSC host exists, too. The MSC host operates as a system to which a MSC device can be attached. Then the MSC host is able to read data from the device as well as to write data on the device.

6.3 Windows Driver for MTP

Windows provides natively the Windows Portable Device drivers to support MTP in Windows 7 and later versions, referring to Section 4. For previous versions of Windows, Microsoft provides the Media Transfer Protocol Porting Kit[6]. This driver supports communication over USB, IP and Bluetooth. Moreover, the WPD driver supports communication with storage devices and music players.

6.4 Object Handles

Object handles represent a logical object on the device, using the *UINT32* type. They will be created on-demand, by establishing a connection. In case of disconnection, the object handles expire and have to be created each time the device will be connected to a host. The device, also responder, is responsible to provide these object handles, allocated to logical objects on the device. An object handle is unique, related to the logical storage of the device.

6.5 Operating Principle

MTP operations occur between an initiator and a responder. In our case, the host is the initiator and the device is the responder. The host initiates all actions with the responder and controls the flow of operations. Furthermore, the initiator enumerates (Section 5.9) the responder. On the other side, the responder only sends responses to operations sent by the initiator. The whole communication is based on what was considered in Section 5.

The following operating principles were logged and analyzed by means of the STM32F429IDISCOVERY development board (Section 3.1) and its MTP responder stack that we put into operation [3].

[6] https://www.microsoft.com/en-in/download/details.aspx?id=19153

6.5.1 MTP Responder

The MTP responder stack on the STM32F429IDISCOVERY (Section 3.1) development board complies with the MTP responder stack described in [6]. As seen in Figure 21, the commands were transmitted over USB. As described in Section 5.4, the endpoints will trigger interrupts to signal incoming data. Then the data will be processed by the MTP responder stack, which is running in a loop. It examines incoming data by means of a lookup table for proper commands or if some specific flags have been set. Such flags indicate if the device is ready to receive new messages or has to prepare further data packets to send. In case of proper commands, which are defined in the MTP Responder Codes [6] and operate as clear commands to initiate activities on the device, the commands will be processed and when necessary interact with the coupled storage to prepare new data for transmission or execution of activities, e.g., deleting, renaming or copying data.

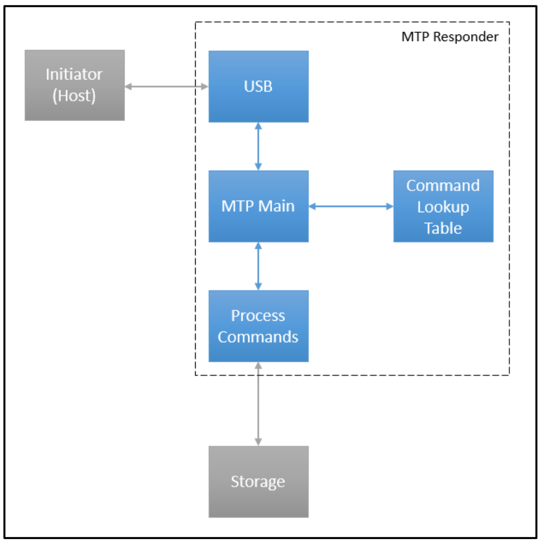


Figure 21: MTP Responder Stack STM32F429IDISCOVERY, inspired by [6]

6.5.2 Read Memory

Figure 22 illustrates how the initiator wants to learn about the content of the device. Usually the initiator wants to know about the device' content after the enumeration (Section 5.9). The command *GetObjectHandles* queries the responder to identify the number of objects and confirm with a MTP_OK packet. Then the initiator wants to know about where the storage directory begins, by sending a *GetObjectPropValue* command. Therefor the responder replies with a data packet, containing the directory in an array with the number of objects. Next the initiator requests the data format of the objects by sending a *GetObjectPropDesc* command. The responder then starts at the beginning of the directory and gathers information about the specific object on the device and replies with the data. Lastly, a *GetObjectInfo* command queries the responder to retrieve the whole binary data and reply to the initiator.

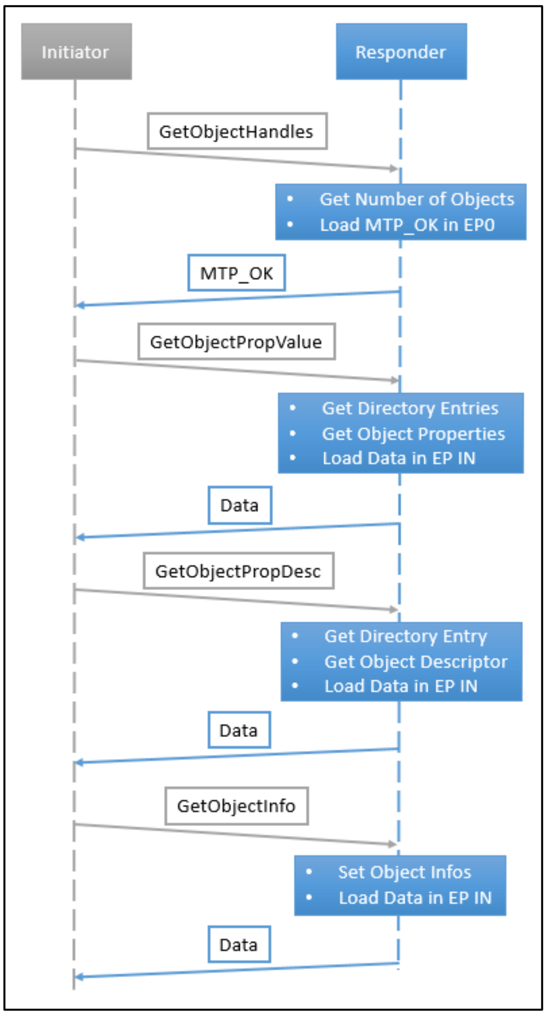


Figure 22: MTP Read Memory Instructions

6.5.3 Delete Objects

As seen in Figure 23, the initiator sends a *DeleteObject* command to delete data on the device. The responder releases the allocated storage and confirm with a MTP_OK packet. Next the initiator sends a *GetStorageInfo* command to learn about a storage area on the device, therefore the responder replies with a data packet containing the information.

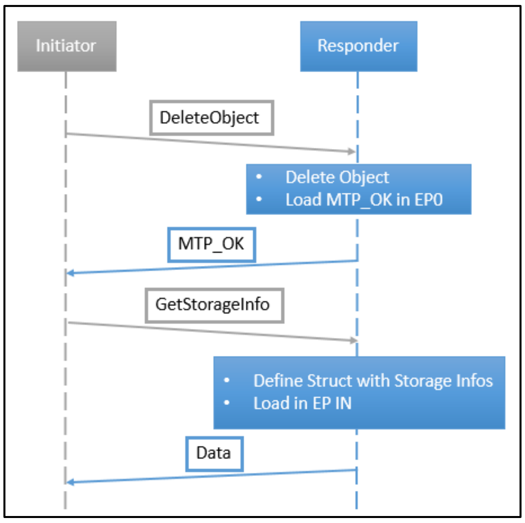


Figure 23: MTP Delete Object Instructions

6.5.4 Rename Objects

First of all, the initiator sends a *SetObjectPropValue* command to indicate that the next data packet will contain the value to change a specific property of an object, as seen in Figure 24. Subsequently the initiator sends the *ObjectPropName*. The responder receives the new object name to change the specific object and executes the operation. To confirm the operation the responder replies with a MTP_OK packet. By means of a *GetObjectPropValue* command, the initiator queries the responder to confirm the change, by sending a packet with the new object name. Lastly, the initiator wants to gather all binary data about the specific object by sending the *GetObjectInfo* command. Then the responder gathers all data and loads them into the EP IN buffer.

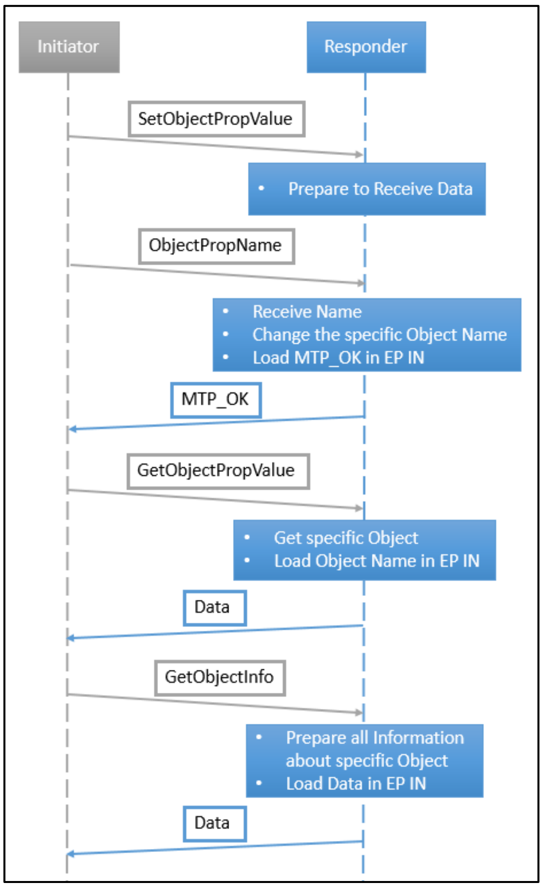


Figure 24: MTP Rename Object Instructions

6.5.5 Download Object from Device

As illustrated in Figure 25, the initiator sends a *GetObject* command to download a specific object from the device. Therefor the responder prepares a data packet and loads the binary data into the EP IN buffer.

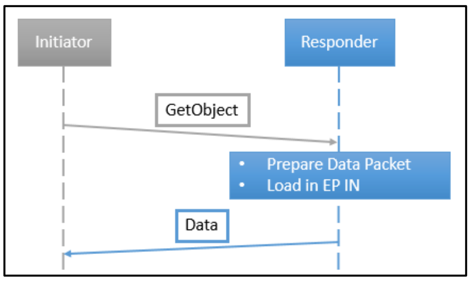


Figure 25: MTP Download Object from Device Instructions

6.5.6 Upload Object to Device

First, the initiator initiates to upload a specific object, by sending a *SendObjectInfo* command. By sending the second *SendObjectInfo* command with object data, the responder creates a new directory as well as a new object to store the information and confirms with a MTP_OK packet. Then, more binary data will be send by the initiator, indicated by *SendObject* command. The binary data will be stored in the new object. The responder confirms with a MTP_OK packet. Furthermore, the initiator sends a *SetObjectPropValue* to change specific properties of the specific object and the responder confirms with a MTP_OK packet again. Lastly, the initiator wants to learn about the uploaded object in the storage area of the device and sends a *GetObjectInfo* command to gather the new information about the object. The responder prepares a data packet with all binary data and loads it into the EP IN buffer.

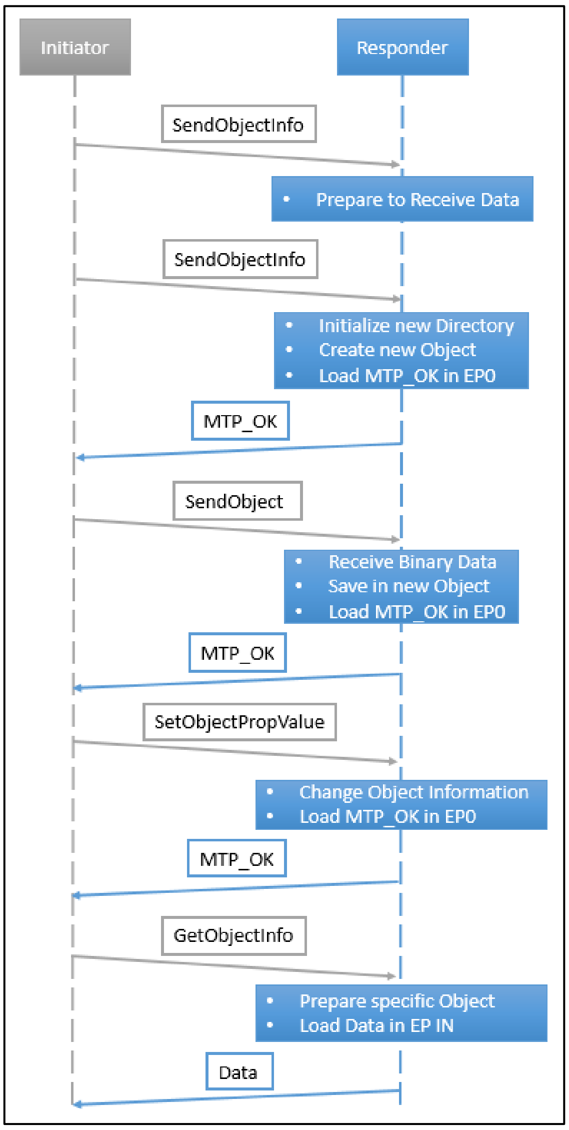


Figure 26: MTP Upload Object to Device Instructions

7 Windows Hardware Certification Kit

The Windows HCK is designed to test different systems, e.g., devices and drivers, based on the Windows Requirements. Succeeded tests guaranty a reliable and compatible system on the operating system which was tested against. To minimize development effort, costs and errors, it would be an advantage to develop hardware, software and drivers among the Windows Requirements concerning the system in development.

While a device is connected to the HCK set-up, it recognizes all available functions. By means of the available functions, the HCK determine which tests must run to verify them. Each test is bound to a requirement that specify which functionality a system must provide to fulfill the standard.

The HCK was designed to verify systems with Windows 7 and 8.1 operating system and is operable on Windows Server releases from Windows Server 2008 R2 to Windows Server 2012[7].

The latest version is called Hardware Lab Kit and provides, besides various updates and improvements, a test environment for Windows 10 and Windows Server 2016 Operating system. We have chosen the HCK because Windows 7 was already available and the development department was developing on Windows 7.

7.1 Test Concept

The Windows HCK test setup consist at least of two components, the HCK test server and one or more test computers. This is required because the test computer is not operable while running tests.

7.1.1 HCK Test Server

It comprises two components. Windows HCK Studio and Windows HCK Manager. The HCK Studio, which is a management tool, is capable to select and run tests on the selected test computer.

The Manager software manages the tests that runs on test computers. A test server can be associated with many test computers.

7.1.2 Test Computer

The test computer is responsible for running the tests, selected by the test server. Each one can be differently configured, e.g., operating system, hardware setups or drivers which depends on the test scenarios. For example, a test computer can be running Windows 10 and a printer is connected to it while another test computer can be running Windows 8 and sets a self-written driver under test. Additionally, a HCK Client, which is stored on the test server, must be installed on the test computer for connectivity and identification purposes. In case of testing devices, the device under test (DUT) must be connected to the test computer. Test computers can only be associated with one test server.

7.1.3 Deployment

The number of computers those are necessary for the setup, depends on the deployment scenario. There are two different deployment scenarios, a domain-joined environment and a workgroup environment.

[7] https://msdn.microsoft.com/en-us/library/windows/hardware/jj124068(v=vs.85).aspx

A domain-joined environment consists at least of three computers. Those are a test server and a test computer which are connected to a domain controller. In a workgroup environment there is no domain controller and it consists at least of two computers. A test server and a test computer which are connected to each other. The used test set-up is built up in a workgroup environment that we will bring into focus in Section 7.2. We proceed with a workgroup environment because it was not necessary to set up the environment on a network to operate with a distant computer. Thus, the workgroup was set up on two computers on one spot. Additionally, it was not necessary to provide the test environment to external offices or other developers.

7.1.4 Test Levels

HCK tests are subdivided in six levels. In Table 5 are four of them mentioned, because level five and six are not essential and just comprise optional tests.

Table 5: HCK Test Levels [7]

Test Level	**Description**
Basic	These are simple and direct tests for developers to run quickly and catch fundamental issues early on. For example, developers are actively working on their device/driver/systems and are in the process of adding functionality. Developers are expected to test their drivers intermittently during development with regression style tests.
Functional	These are feature level test. At this stage, all functionality should be complete. For example, partners are required to run more rigorous testing at this stage to test the full functionality of their device/driver/system. If their device/driver/system passes all feature tests, then the device/driver/system is considered *feature-complete*.
Reliability	These are stress level test. These tests may require special setup and requirements. At this stage, your device/driver/systems are fully functional and should be tested under stress scenarios. The goal here is to ensure the reliability of the drivers/devices/systems.
Certification	These are all test required for Windows Certification. At this stage, you are ready to submit your drivers/devices/systems for certification. If you passed all previous levels (basic, functional, reliability), this stage should be simple.

7.2 Test Environment

Two computers were connected directly via an Ethernet crossover cable, which will be used to connect the same type of devices. The DUT was attached to the test computer via micro USB 2.0 (Figure 27).

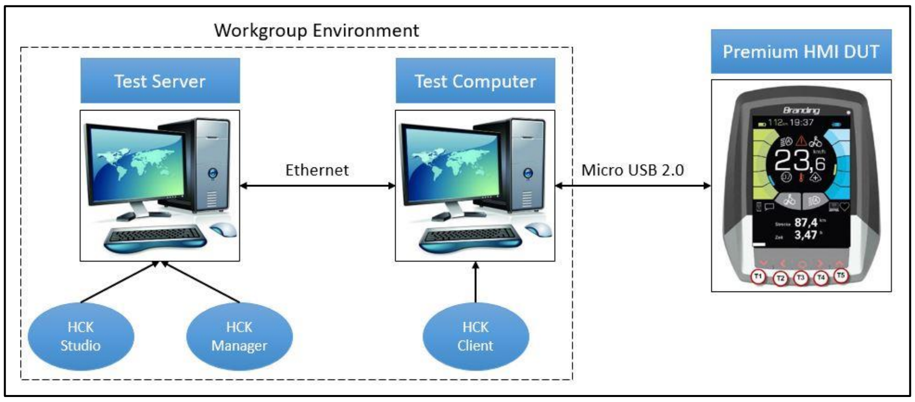


Figure 27: HCK Test Environment

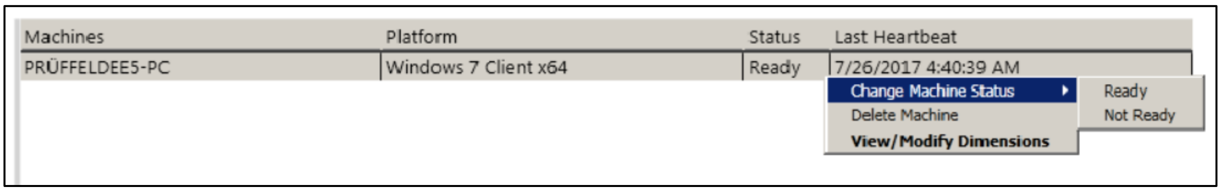
The task concerning the test environment, comprise to install and set up the HCK test server, HCK Studio, and HCK Manager and on the other side the HCK Client on the test computer.

7.2.1 HCK Test Server and HCK Client

The HCK test server is set up on a machine running Windows Server 2008 R2 Standard 64-Bit that runs HCK Studio and HCK Manager. On the other side the test computer is running Windows 7 Professional 64-Bit that runs the HCK Client. It is necessary that the operating system running on the HCK test server is a version of Windows Server. As mentioned before in Section 7, depending on the Windows Server version, there are more or less compatible Windows versions to test against.

7.2.2 HCK Studio

The HCK Studio and HCK Manager can be downloaded from Microsoft[8] directly from the test server. To install the programs, it is necessary and important that the operating system is running the English version of Windows Server. Within the machine pool in HCK Studio, the test computer should be visible and in the *Ready* state for interaction as seen in Figure 28.

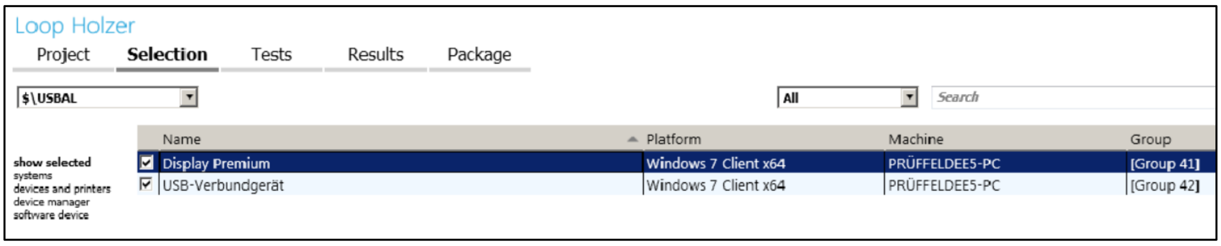


Machines	Platform	Status	Last Heartbeat
PRÜFFELDEE5-PC	Windows 7 Client x64	Ready	7/26/2017 4:40:39 AM

Figure 28: HCK Studio Machine State

[8] https://developer.microsoft.com/en-us/windows/hardware/windows-hardware-lab-kit

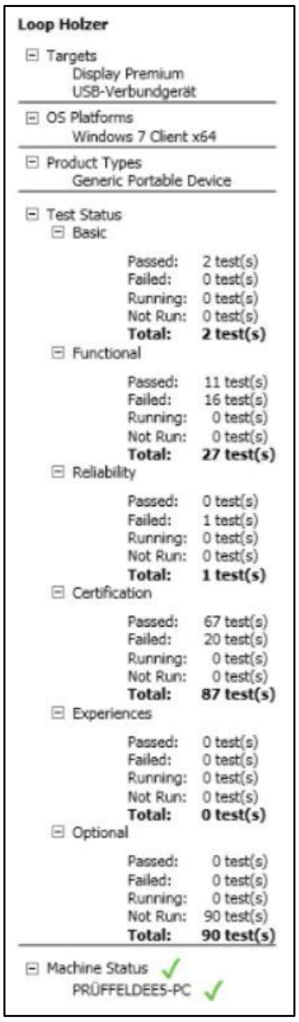
Besides the default machine pool, it is possible to create additional machine pools. Each test computer can only be assigned to one machine pool. The first step, to initiate a test, is to create a project with a specific name. A test computer can be assigned in more than one project. Second, a list from the test computer, with attached hardware and their assigned driver become available, to select which system should be tested. Figure 29 shows the project *Loop Holzer* that concerns the DUT. *Display Premium* and *USB-Verbundgerät* were selected and concerning MTP and USB related to the DUT.



	Name	Platform	Machine	Group
✓	Display Premium	Windows 7 Client x64	PRÜFFELDEE5-PC	[Group 41]
✓	USB-Verbundgerät	Windows 7 Client x64	PRÜFFELDEE5-PC	[Group 42]

Figure 29: HCK Studio Available Systems

In the *Tests* tab, all tests for a selected system will be listed. By running one or more tests, they will get into a schedule and assigned to all machines that are included in the machine pool. Moreover, the tests can be separated by selecting a specific test level. The Graphical User Interface (GUI) comprise current information about the tests and the test computer, the current system settings and the test status as seen in Figure 30.



Loop Holzer

Targets
Display Premium
USB-Verbundgerät

OS Platforms
Windows 7 Client x64

Product Types
Generic Portable Device

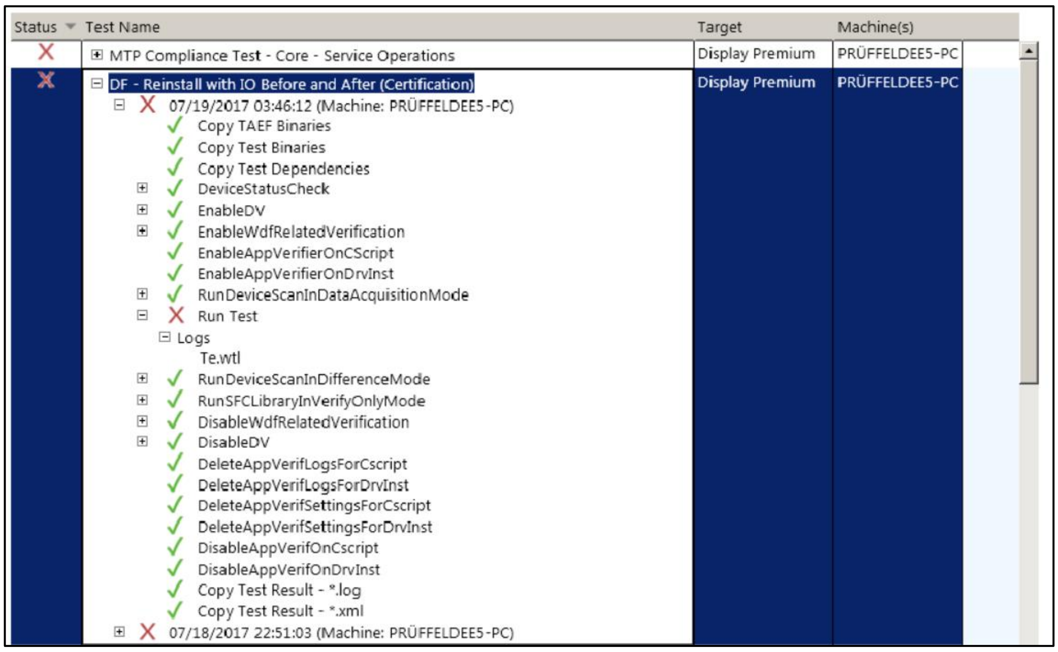
Test Status

	Basic	Functional	Reliability	Certification	Experiences	Optional
Passed:	2 test(s)	11 test(s)	0 test(s)	67 test(s)	0 test(s)	0 test(s)
Failed:	0 test(s)	16 test(s)	1 test(s)	20 test(s)	0 test(s)	0 test(s)
Running:	0 test(s)	0 test(s)	0 test(s)	0 test(s)	0 test(s)	0 test(s)
Not Run:	0 test(s)	0 test(s)	0 test(s)	0 test(s)	0 test(s)	90 test(s)
Total:	**2 test(s)**	**27 test(s)**	**1 test(s)**	**87 test(s)**	**0 test(s)**	**90 test(s)**

Machine Status ✓
PRÜFFELDEE5-PC ✓

Figure 30: HCK Studio Test Status & Client

Furthermore, when a test completes, the results are shown in the *Results* tab. Each test case can be opened up and contains a listed view about each time the test was running. Inside each test case is a detailed view about the test sequence, information if the test has passed or failed and a created log with detailed information about the results, referring to.



Status	Test Name	Target	Machine(s)
✗	⊞ MTP Compliance Test - Core - Service Operations	Display Premium	PRÜFFELDEE5-PC
✗	⊟ DF - Reinstall with IO Before and After (Certification)	Display Premium	PRÜFFELDEE5-PC

```
⊟ ✗ 07/19/2017 03:46:12 (Machine: PRÜFFELDEE5-PC)
      ✓ Copy TAEF Binaries
      ✓ Copy Test Binaries
      ✓ Copy Test Dependencies
    ⊞ ✓ DeviceStatusCheck
    ⊞ ✓ EnableDV
    ⊞ ✓ EnableWdfRelatedVerification
      ✓ EnableAppVerifierOnCScript
      ✓ EnableAppVerifierOnDrvInst
    ⊞ ✓ RunDeviceScanInDataAcquisitionMode
    ⊟ ✗ Run Test
       ⊟ Logs
          Te.wtl
    ⊞ ✓ RunDeviceScanInDifferenceMode
    ⊞ ✓ RunSFCLibraryInVerifyOnlyMode
    ⊞ ✓ DisableWdfRelatedVerification
    ⊞ ✓ DisableDV
      ✓ DeleteAppVerifLogsForCscript
      ✓ DeleteAppVerifLogsForDrvInst
      ✓ DeleteAppVerifSettingsForCscript
      ✓ DeleteAppVerifSettingsForDrvInst
      ✓ DisableAppVerifOnCscript
      ✓ DisableAppVerifOnDrvInst
      ✓ Copy Test Result - *.log
      ✓ Copy Test Result - *.xml
⊞ ✗ 07/18/2017 22:51:03 (Machine: PRÜFFELDEE5-PC)
```

Figure 31: HCK Studio Test Results Tab

All in all, the HCK Studio software is only the GUI to manage the machine pool and control which test is supposed to run on a machine pool.

7.2.3 HCK Manager

On the other side, the HCK Manager takes care of the operable part concerning the HCK test server. First of all, the Manager collects all information from the test computer about systems and attached devices. It is responsible for scheduling the executed tests by HCK Studio and distribution of each test computer in a machine pool. Second, the tests, which are referred to as jobs, will be sending to the test computer. Its GUI provides the functionality to handle already scheduled jobs or create new jobs. Otherwise the Manager will be triggered from the Studio to execute selected tests. The test computer collects the results of a job in a log and sends it back to the Manager on the test server after the job completed (Figure 32).

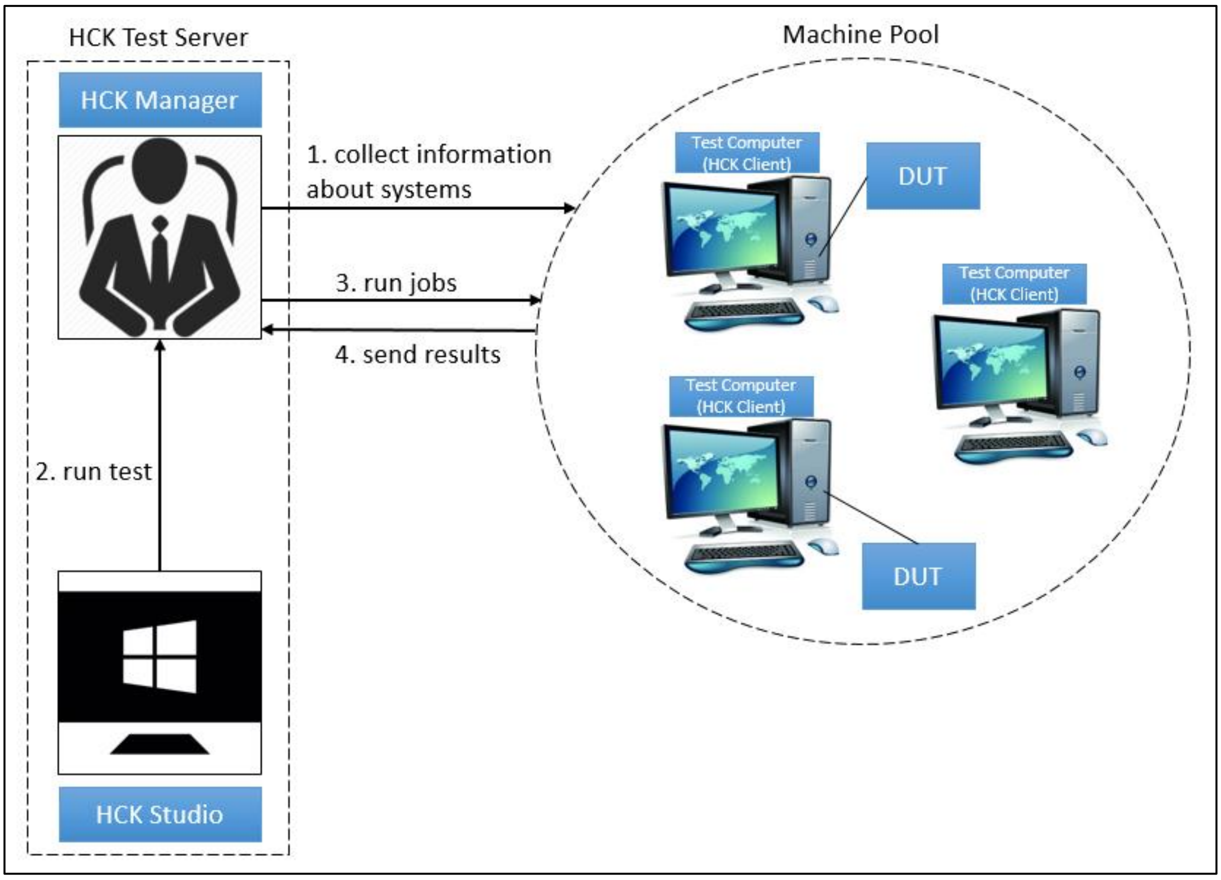

HCK Test Server
Machine Pool
HCK Manager
1. collect information about systems
3. run jobs
4. send results
2. run test
HCK Studio
Test Computer (HCK Client)
DUT
Test Computer (HCK Client)
Test Computer (HCK Client)
DUT

Figure 32: HCK Manager Operating Principle

Finally, the whole test process in the logs can be analyzed within the HCK Manager. They can be accessed through the HCK Studio as mentioned before in Section 7.2.2. Nevertheless, it is not necessary to interact directly with the Manager, in case of selecting and executing tests.

7.2.4 HCK Client

The HCK client is going to be installed from a shared network location which is located on the test server. The location path is *\\<NameOfYourTestServer>\HCKInstall\Client\Setup.exe* and can be called directly from the test computer. After the HCK Client installed successfully, the test computer is going to be visible in the HCK Studio and in *Ready* state as seen in Figure 28.

7.2.5 Windows Server

"Windows Server is a group of operating systems designed by Microsoft that supports enterprise-level management, data storage, applications, and communications. Previous versions of Windows Server have focused on stability, security, networking, and various improvements to the file system. Other improvements also have included improvements to deployment technologies, as well as increased hardware support." [8]

7.2.6 DUT

The DUT is the HMI device for *Pedelecs* from *Marquardt GmbH*. It displays driver information e.g. current speed, statistics, driven kilometers, remaining distance and provides configuration options e.g. reset statistics, language or unit settings etc. through a 3.5" TFT Full Color Display [9]. It is connected via micro USB 2.0 to the test computer and in addition it provides the USB protocol MTP that is implemented as an USB custom class. Having taken these factors into account, the objectives are to set USB and MTP under test in Windows HCK.

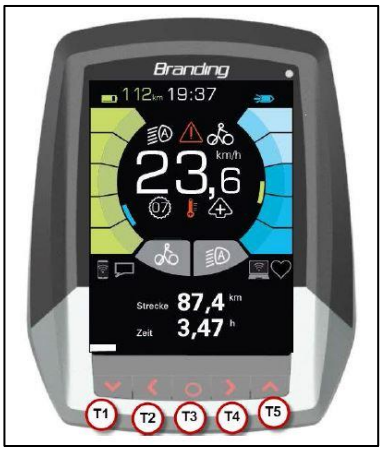


Figure 33: HMI Device Marquardt GmbH

7.3 Hardware Test Categories

The Windows Certification Program provides many tests for any feature, associated with devices attached to the test computer. Those tests are divided in six different levels and organized into hardware test categories. In case of the objective relating to this chapter, verification of USB and MTP, we will only discuss the hardware test cases for USB and MTP.

7.3.1 Device Connectivity Tests

These should be the first tests before starting to test any other features of the DUT. Without a constant connection, nothing else can be tested. All in all, there are nine of 30 tests associated to the DUT. Therefore, the test cases examine the DUT if it responds properly to an unique assigned address. Furthermore, the standard descriptor data respond is going to be checked and some stress tests, i.e., Enumeration, disable and enable operations under permanent load.

7.3.2 Device Fundamentals Tests

In general, this category tests device fundamentals. In total there are five specific tests which tests the fundamentals of USB and MTP. The tests verify the proper implementation and functionality of UMDF and KMDF drivers (Section 4) concerning MTP.

7.3.3 Device Fundamentals Reliability Tests

This category belongs to the Device Fundamentals Tests category. It includes 48 tests therefrom 24 tests which fits in our test set-up. General I/O operations are going to be executed to verify that the DUT was properly configured. This comprise operations like rebooting, disabling and enabling the device.

7.3.4 Device Portable Tests

To run these tests the DUT must appear in the Device Manager under Portable Devices. It is necessary that the Windows Class Driver for MTP is going to be used. In case of using a custom INF file (Section 4.1) related to the DUT, the test is going to lead to a failure.

The MTP implementation on the DUT will be tested, to see if it is compliant with the Windows implementation of MTP. When a device will be connected or disconnected through Plug and Play, the Windows Media Device Manager (WMDM) application is going to start and updates the cache device list with connected devices. The tests examine WMDM scenarios, i.e., content transfer, device initialization and content cancelation for Portable Media Devices. Other tests verifying that the DUT fulfills the Windows Hardware Certification requirements, which must work properly with the Windows Portable Device API. This also includes specific MTP functionality tests, e.g., *MultiSessions*.

7.4 HMI Device Test Analysis

The successfully executed test cases confirm that the device provides read and write operations. In detail, the Device Portable Tests confirm that the device and device driver in accordance with the WPD scenarios operate properly at the WPD API level which comprise operations to explore a device, send and receive content.

Furthermore, the Device Connectivity Tests that were executed successfully, verify that the device responds properly to descriptor requests (Section 5.7) that contains correct data. Due to that, the enumeration (Section 5.9) runs properly, verified by these tests, too.

Last, the Device Fundamentals and Device Fundamentals Reliability Tests, referred as DF in Table 6, that were executed successfully, verify that Plug and Play operations do not lead to errors and fundamental communication on WDF (Section 4) level will be performed properly

To consider the failed tests, we are going to explain the test case and lastly a solution that helps to run the test successfully. To get an overview, the tests are going to be retrievable in Table 6 with detailed information.

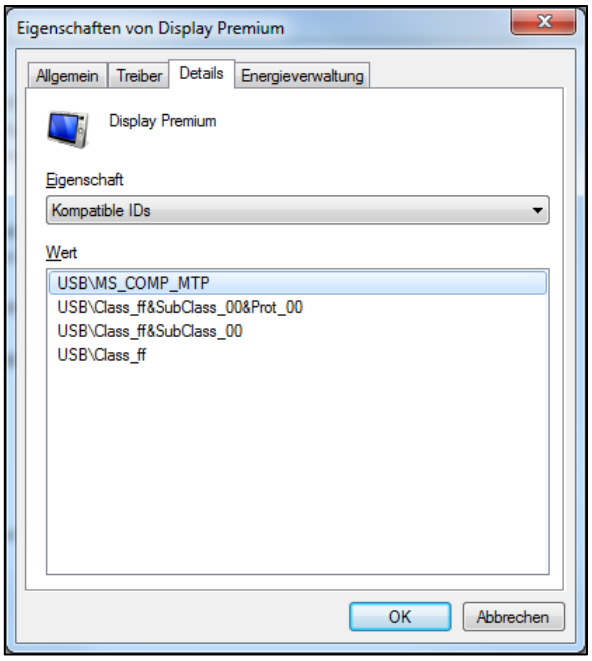
Table 6: HMI Device List of Failed Tests

Test Name	Hardware Category	Target	Test Level
DF - Reinstall with IO Before and After	Device Fundamental Reliability	MTP	Certification
WDF - Check KMDF Coinstaller Version Test	Device Fundamentals	MTP	Certification Functional
WDF - Check UMDF Coinstaller Version Test	Device Fundamentals	MTP	Certification Functional
USB-IF Certification Validation Test (Device)	Device Fundamentals	USB	Certification
USB Serial Number	Device Connectivity	USB	Certification
WPD Compliance Tests - Events (Manual)	Device Portable	MTP	Certification Functional
WPD Compliance Stress Tests	Device Portable	MTP	Certification Reliability
MTP Compliance Test - Core - Device Properties	Device Portable	MTP	Certification Functional

Test Name	Hardware Category	Target	Test Level
MTP Compliance Test - Core - Miscellaneous	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Object Properties	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Operations	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Plays For Sure	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Recommended	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Service Fundamentals	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Service Identification	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Service Miscellaneous	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Service Operations	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Transport	Device Portable	MTP	Certification Functional
MTP Compliance Test - Core - Unsupported	Device Portable	MTP	Certification Functional
MTP Compliance Test - Requirements - Core	Device Portable	MTP	Certification Functional

7.4.1 DF – Reinstall with IO Before and After

In this test scenario the test computer is going to uninstall and reinstall the driver for the DUT to verify that no other device or driver is going to be affected. When the test starts, a *preinstallscan.xml* file will be created and collects all the information about connected devices before the test process starts. When completing the test, a *postinstallscan.xml* file will be created and collects all the information about connected devices. The two files will be compared, in order to examine if the reinstallation process had any effect on devices or drivers. A failed test result represents that there is a difference between the *preinstallscan.xml* and *postinstallscan.xml* files. The shown error message in the report *WDTF_DRIVER_SETUP_DEVICE : DriverSetupDevice::Install() call class installer HRESULT=0x800F0228* describes, that there are no existing compatible drivers for the DUT. The test result is not really surprisingly because the DUT will not be automatically recognized by Windows. This is based on the false USB class code that will be provide in the configuration descriptor from the device as seen in Figure 34. The valid class codes were mentioned in Section 4.1.



Eigenschaft: Kompatible IDs

Wert:

USB\MS_COMP_MTP
USB\Class_ff&SubClass_00&Prot_00
USB\Class_ff&SubClass_00
USB\Class_ff

Figure 34: Device Manager, USB Class Codes from DUT

But how can we deal with this issue? The DUT was made operable by setting up Windows to handle the device as a MTP device over the Device Manager. But still the class codes in software should be changed, instead it is necessary to provide a custom INF file to allocate the correct driver. Moreover, the created files, *preinstallscan.xml* and *postinstallscan.xml*, were both equal to each other, thus the case of difference of the files can be excluded.

7.4.2 WDF – Check UMDF Coinstaller Version Test

The test ensures that the drivers, assigned to the DUT, are signed. While the executed job was looking for the device with VID: 0421 and PID: 0485 an error occurred. In fact, this is exactly what should happen because the DUT driver was not installed properly, as mentioned in Section 7.4.1. The DUT has the VID: 28FE and PID: 1125. But where does the other ID's come from?

Since the mentioned install procedure, the MTP driver is related to the standard INF file from Windows that contains the given ID's.

7.4.3 WDF – Check KMDF Coinstaller Version Test

See Section 7.4.2 WDF – Check UmdF Coinstaller Version Test.

7.4.4 USB Serial Number

Some devices must provide an unique serial number that will be tested, e.g., Bluetooth or mass storage devices, except MTP. All in all, this is supposed to fail.

7.4.5 WPD – Compliance Tests – Events (Manual)

The test forces the DUT to reset, to verify that the device works properly with the WPD stack. It is important that the DUT is now allowed to be unplugged or turned off.

If the DUT is unplugged or turned off, a *WPD_EVENT_DEVICE_REMOVED* event is triggered. To run the test successfully, the test computer expects a *WPD_EVENT_DEVICE_RESET* event instead. To sum up, a reset event should not be confused with a hardware or power-on reset that pulls down V-Bus power. Instead a reset pulls down D- and D+ pin (Section 5.2) for ten milliseconds, which initiate the enumeration procedure.

As a result, the test failed. Therefore, a software reset implementation is not implemented and the DUT is not able to execute a reset.

7.4.6 WPD – Compliance Stress Tests

By exercising the driver, this test verifies that the driver complies with the Windows Hardware Certification requirements. Device and driver are going through different scenarios, e.g., object hierarchy enumeration (Section 5.9), object deletion, object resource operation and power management. Unlike to the test results concerning other tests, there is no log file that helps to analyze the test scenario and find the errors.

Therefore, we can only assume what went wrong, due to the proper functionalities. Since the enumeration process on the test computer works properly and a test case, called *USB Enumeration Stress (Win7)* [9], was completed successfully, we can exclude a failure in the enumeration process. Furthermore, the test forces the transmission of object properties that works fine on the test computer. Then, during the test process, objects are written to the device and read from device. A failure in this process is unlikely, since this is working properly by manual interaction with the DUT. Next, deletion of objects works properly, too.

In conclusion, we can narrow down the source of errors to content type and format requirements. In detail, this means that possibly some content types and formats are not supported, which are listed below.

- WPD_CONTENT_TYPE_AUDIO (Describes the general type)
- WPD_OBJECT_FORMAT_WMA (Windows Media Audio format)
- WPD_OBJECT_FORMAT_MP3 (Audio format)
- WPD_OBJECT_FORMAT_WMV (Windows Media Video format)
- WPD_OBJECT_FORMAT_BMP (Windows Bitmap, image format)
- WPD_OBJECT_FORMAT_EXIF (Exchangeable File Format, image format)

[9] The USB Enumeration Stress (Win7) test enumerates the DUT n times.

- WPD_OBJECT_FORMAT_GIF (Graphics Interchange Format, image format)
- WPD_OBJECT_FORMAT_ICON (Windows Icon format)
- WPD_OBJECT_FORMAT_JFIF (JPEG Interchange Format, image format)
- WPD_OBJECT_FORMAT_JP2 (JPEG2000 Baseline File Format, image format)
- WPD_OBJECT_FORMAT_JPX (JPEG2000 Extended File Format, image format)
- WPD_OBJECT_FORMAT_PNG (Portable Network Graphics, image format)
- WPD_OBJECT_FORMAT_TIFF (Tag Image File Format, image format)

7.4.7 MTP Compliance Test – Core – Service Operations

This test ensures that the DUT is using the MTP class driver and verifies compliance with the Windows implementation of MTP. The DUT will be recognized by the test computer as an active MTP device with VID: 28FE and PID: 1125.

The test server holds a *mtptest.inf* file which contains the ID's assigned to the test devices. While trying to update this file, the process failed. In this case the INF file, related to the driver, does not contain the VID and PID of the device that was recognized by the enumeration process and leads to an error. Consequently, this is based on what was mentioned in Section 7.4.1 because Windows was prompted to refer its INF file. Due to the error code 0x103, which means that "No more data is available." [10], it seems applicable.

The following tests are included in the same hardware test category and failed for the same reason. In case of a lack of a description for each followed test in HCK it is only possible to list them.

- MTP Compliance Test – Core – Transport
- MTP Compliance Test – Core - Service Miscellaneous
- MTP Compliance Test – Core - Service Identification
- MTP Compliance Test – Core - Service Fundamentals
- MTP Compliance Test – Core - Plays For Sure
- MTP Compliance Test – Core - Operations
- MTP Compliance Test – Core - Object Properties
- MTP Compliance Test – Core - Miscellaneous
- MTP Compliance Test – Core - Recommended
- MTP Compliance Test – Core - Unsupported
- MTP Compliance Test – Core - Device Properties

7.4.8 MTP Compliance Test – Requirements – Core

The test verifies if core requirements for MTP devices are supported.

See Section 7.4.7 MTP Compliance Test – Core – Service Operations.

8 Conclusion

As a result, the USB stack, as well as the MTP responder stack, were clarified and explained in their function. Illustrations clarify various procedures to understand the different functions.

First, we documented the concept of Windows driver, because our evaluation platform is Windows 7. Moreover, this is fundamental to understand how Windows recognizes USB applications. Without drivers we are not able to use them.

Furthermore, we analyzed the USB stack and pointed out that all data between the host and a device will be transmitted through endpoints. They serve as buffer, which are used from the host and the device. They are placed on the device as hardware and were driven from both. The host is using them to send data to the device and vice versa, but in detail the function of the device only writes and reads data on the buffer. Basically, the transmission will be initiate by the hardware itself. There are also various transfer types for different applications, for example transferring sensitive data or just media. In the end the enumeration is illustrated that explains how Windows detects an attached device to load an appropriate driver.

Based on USB, the MTP responder stack was analyzed, too. MTP applications need several drivers to be recognized on an operating system, which are natively on Windows 7. Additionally, we discussed the main differences of MTP and MSC and pointed out the big difference. By means of the STM32F429IDSCOVERY development board, which was running the MTP application, we achieved to analyze the operating principle and illustrated the processes in a clear manner.

Lastly, on the basis of the achieved knowledge about USB and MTP, a test environment, based on Windows Hardware Certification Kit, was set up, with the objective to set a HMI device, from *Marquardt Verwaltungs GmbH*, under test. All test cases are predefined and directly executable without making many modifications. Finally, the test cases were successfully executed and subsequently their results were analyzed. We were able to test a device to find out what was missing and what operated properly, in case of USB and MTP. By considering the successfully executed tests, it is verified that the device provides read and write operations. Descriptors contain correct data and were transmitted in case of descriptor requests properly, which lead additionally to a proper enumeration. The failed tests prove that the DUT has a lack of appropriate drivers. Windows is not able to install the appropriate driver because the DUT delivers the wrong USB class codes, during the enumeration, which do not comply with the MTP class codes. Next the DUT's software is not able to perform a software reset to reinitiate the enumeration, without pulling the bus power low. Perhaps the DUT does not support some specific file formats, but in case of a non-existing log file, regarding the specific test, it is not possible to verify why the test failed. It was only possible to narrow down the source of errors.

Subsequent works could use this thesis to get more in-depth with the MTP responder stack. They could start to analyze the detailed data transfer, by means of the appended MTP responder stack for the STM32F429IDISCOVERY development board and try to figure out how data packets are built up in detail. In the end a MTP implementation could be developed from scratch.

If the Windows Hardware Certification Kit is going to be used by companies to verify their devices and systems, it would be recommendable to automate the testing, in case of a greater extent. This would be possible by using written scripts to accelerate specific test cases for lots of devices and systems.

9 Summary

The concrete task was to analyze the USB stack as well as the MTP responder stack by means of the STM32F429IDISCOVERY development board. Concerning this matter, both had to be clarified in a clear manner and emphasized by illustrations. Then a test environment had to be set up, which allows to set an USB device, running a MTP responder stack, under test. The test results had to be evaluated.

By means of the USB MTP application, running on the development board, and additional placed UART outputs between the code lines, it was easier to understand the processes, while the application was operating. It was necessary to avoid to place much outputs, because the USB communication must be executed in a specific time. For the test environment the Windows Server 2008 R2 operating system had to be set up to run the Windows Hardware Certification Kit. Then predefined test cases were able to be executed and to evaluate the subsequent test results.

The outcome is an overview about the USB stack and MTP responder stack and emphasized by illustrations to help to understand the operating principle of both. A test environment was set up with a detailed description. We were able to test a HMI device successfully and to evaluate the test results. In conclusion, then, it is clear that the HMI device test results reveal that they are based on driver issues, because the device is using an incorrect USB class specification code. This causes that the host is not able to recognize the DUT and to select an appropriate driver.

Bibliography

[1] Teledyne LeCroy, "Product Analyzers USB," w. D. w. M. w. Y.. [Online]. Available: http://teledynelecroy.com/protocolanalyzer/protocoloverview.aspx?seriesid=414&capid=103&mid=511. [Accessed 01 August 2017].

[2] STMicroelectronics, "UM1734 User Manuel, STM32Cube USB device library," w. D. May 2015. [Online]. Available: http://www.st.com/content/ccc/resource/technical/document/user_manual/cf/38/e5/b5/dd/1d/4c/09/DM00108129.pdf/files/DM00108129.pdf/jcr:content/translations/en.DM00108129.pdf. [Accessed 26 August 2017].

[3] Sebastian, "MTP Responder Stack for STM32F429IDISCOVERY," [Online]. Available: private communication. [Accessed 15 May 2017].

[4] C. Peacock, "USB Functions, Beyond Logic," 17 September 2010. [Online]. Available: http://www.beyondlogic.org/usbnutshell/usb3.shtml#Endpoints. [Accessed 17 May 2017].

[5] C. Peacock, "The Setup Packet, Standard Device Requests, Beyond Logic," 24 June 2011. [Online]. Available: http://www.beyondlogic.org/usbnutshell/usb6.shtml. [Accessed 22 June 2017].

[6] Microsoft, "download.microsoft.com," w. D. January 2012. [Online]. Available: https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0ahUKEwim1abj4tPVAhUGJ8AKHeTBDyEQFgg3MAI&url=http%3A%2F%2Fdownload.microsoft.com%2Fdownload%2F2%2F4%2FA%2F24A36661-A629-4CE6-A615-6B2910A1367A%2FInside%2520MTP%2520Responder.pdf&usg=AFQjCN. [Accessed 13 July 2017].

[7] Microsoft, "HCK Testing Concepts, Microsoft Developer Network," Microsoft Coorperation, w. D. w. M. w. Y.. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/hh998767(v=vs.85).aspx. [Accessed 02 July 2017].

[8] Microsoft, "Windows Server, Microsoft Developer Network," w. D. w. M. w. Y.. [Online]. Available: https://msdn.microsoft.com/en-us/library/dn636873(v=vs.85).aspx. [Accessed 26 July 2017].

[9] Marquardt GmbH, "Marquardt," w. D. w. M. w. Y.. [Online]. Available: https://www.marquardt.com/download/datenblaetter.html. [Accessed 27 July 2017].

[10] Microsoft, "System Error Codes, Microsoft Developer Network," Microsoft, w. D. w. M. w. Y.. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681382(v=vs.85).aspx. [Accessed 21 July 2017].