LIÈGE université Library

https://lib.uliege.be

https://matheo.uliege.be

Implementation of video streaming via USB on a microcontroller

Auteur : Rousseau, Alex
Promoteur(s) : Boigelot, Bernard
Faculté : Faculté des Sciences appliquées
Diplôme : Master : ingénieur civil électricien, à finalité spécialisée en "electronic systems and devices"
Année académique : 2024-2025
URI/URL : http://hdl.handle.net/2268.2/23295

Avertissement à l'attention des usagers :

Tous les documents placés en accès ouvert sur le site le site MatheO sont protégés par le droit d'auteur. Conformément aux principes énoncés par la "Budapest Open Access Initiative"(BOAI, 2002), l'utilisateur du site peut lire, télécharger, copier, transmettre, imprimer, chercher ou faire un lien vers le texte intégral de ces documents, les disséquer pour les indexer, s'en servir de données pour un logiciel, ou s'en servir à toute autre fin légale (ou prévue par la réglementation relative au droit d'auteur). Toute utilisation du document à des fins commerciales est strictement interdite.

Par ailleurs, l'utilisateur s'engage à respecter les droits moraux de l'auteur, principalement le droit à l'intégrité de l'oeuvre et le droit de paternité et ce dans toute utilisation que l'utilisateur entreprend. Ainsi, à titre d'exemple, lorsqu'il reproduira un document par extrait ou dans son intégralité, l'utilisateur citera de manière complète les sources telles que mentionnées ci-dessus. Toute utilisation non explicitement autorisée ci-avant (telle que par exemple, la modification du document ou son résumé) nécessite l'autorisation préalable et expresse des auteurs ou de leurs ayants droit.

University of Liège
Faculty of Applied Science

Implementation of video streaming via USB on a microcontroller

Alex **ROUSSEAU**

Thesis presented to obtain the degree of :
Master of Science in Electrical Engineering

Thesis supervisor :
Bernard **BOIGELOT**

Internship supervisor :
Alex **ROSCA**
at *Get Your Way*

Academic year: **2024 - 2025**

Abstract

Implementation of video streaming via USB on a microcontroller

Alex **Rousseau**

Master of Science in Electrical Engineering
University of Liége - Academic year: 2024 - 2025

Thesis supervisor : Bernard **BOIGELOT**
Internship supervisor : Alex **ROSCA** at Get Your Way

The company *Get Your Way* has developed a lightweight Head Mounted Display (HMD) device that is able to display a graphical interface. The goal of this Master's thesis is to develop a firmware that implements the reception of a video stream through USB-C, and the display of said video stream on the HMD screen.

The main task is the implementation of a USB Video Class (UVC) driver for the RZ/A2M microcontroller present inside the HMD. A USB 2.0 host driver library was already implemented but had to be modified extensively to support the UVC driver. To that end, the USB 2.0, UVC 1.0 and EHCI standards were studied and are detailed in depth in this thesis.

The decoding and display of video frames was implemented by integrating relevant sections of sample codes provided by the company which designed the microcontroller, Renesas. The description of these modules are thus less detailed than for the UVC driver.

The firmware was first implemented on a a development board, for easier debugging. It was then supposed to be ported to the HMD board, but a hardware limitation was found. As the new board was not ready at the time of writing this thesis, the porting process was only described and not carried out.

A state of the art of the physical interfaces which allow video streaming was also performed, with a focus on those commonly found in cameras. The intent was to determine the best interface for a possible new version of the HMD, as well as broaden personal knowledge.

Declaration on the use of automatic tools for writing the manuscript

I hereby certify that I have not used any generative intelligence tool in the writing of text, graphics, images, or data reproduced in this manuscript.

The only automatic tool used to write this document is a spelling checker integrated in the text editor.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface.

CPU Central Processing Unit.

CS Control Selector.

DMA Direct Memory Access.

EHCI Enhanced Host Controller Interface.

EMI ElectroMagnetic Interference.

FSM Finite State Machine.

H/W Hardware.

HCD Host Control Driver.

HCI Host Controller Interface.

HMD Head-Mounted Display.

iTD Isochronous Transaction Descriptor.

JCU JPEG Codec Unit.

LSI Large Scale Integration.

MGR Host Manager.

MPS Maximum Packet Size.

OTG On-The-Go.

QH Queue Head.

qTD Queue element Transfer Descriptor.

RTOS Real-Time Operating System.

SerDes Serializer/Deserializer.

TMDS Transition Minimized Differential Signaling.

USB Universal Serial Bus.

UVC USB Video Class.

VC VideoControl.

VDC Video Display Controller.

VS VideoStreaming.

Chapter 1

Introduction

1.1 Get Your Way

1.1.1 The Company

Get Your Way [1] is a young technology company founded in 2020 by 3 engineers from the university of Liège. Their main product is a lightweight Head-Mounted Display (HMD) called *arDent Display*. It started off as a graduation project, intended to be used to display route indications for cyclists. As the company matured, it specialized into providing solutions to the picking [1] issue, for the hospital and logistics sectors.

1.1.2 arDent Display

The *arDent Display*, shown on figure 1.1, is composed of two parts which interlock, attached to a headband. The first part contains the screen and a microcontroller, whereas the other contains a battery, as well as the only cable receptacle: a USB-C port. This USB-C port currently has two uses: battery charge, and data transfer. Indeed, the device can be used as as mass storage peripheral when plugged to a PC, which allow convenient data transfer at a relatively high speed. It also contains a Bluetooth module, which allows transfers of small amounts of data without any cable.

The operator can not interact with the *arDent Display* directly: the only button serves to power on and off the device. However, interaction is possible using an intermediate device connected to it via Bluetooth. For instance, a phone or *Get Your Way*'s wrist-attached keypad: *ScanPad*.

The battery module is removable and two are provided with the device. This allows to charge one battery while the other is being used. A battery lasts 8 hours of operation

[1] The retrieval of items from a storage.

Figure 1.1: *arDent Display*
Source: [2]

1.2 Objective

Get Your Way wants to add the capability to display video on the *arDent Display*. The possible use-cases are for instance, to connect an endoscope in order to look in places which an operator can not reach or fit in, or take photos using the video stream display on the screen to aim. Since Bluetooth has a low data rate (up to 3Mb/s [3]), using the USB-C physical interface is preferable. However, *Get Your Way* does not possess in-house expertise about the USB standard as they have used contractors for USB-related features, such as the mass storage peripheral functionality. This is why this project was proposed as a master's thesis.

The project thus consists in implementing the reception of a video stream, and display of received video frames on the *arDent Display*.

1.3 Thesis Organization

The rest of this document is organized into different chapters, described successively in this section.

Chapter 2 contains a state of the art about video formats and physical interfaces related to video transfer. This analysis ignores the *arDent Display* hardware, in order to develop a broad summary of existing technologies. The intent is both to provide *Get Your Way* information to help them design a possible future version of *arDent Display*, and to extend personal knowledge.

The two following chapters, 3 and 4, provide in depth summaries about the USB 2.0 and USB Video Class (UVC) standards, detailing every concept needed to understand the drivers used by the video streaming application. Chapter 5 details the standardized interface between a USB hardware module and driver for USB 2.0, the EHCI.

Thereafter, the implementation process of the video streaming application is described in chapter 6. This implementation was performed on a development board, which allows easier debugging. Furthermore, sample applications designed for this board can conveniently be tested before porting them to the target device: the *arDent Display*.

The implementation process on the *arDent Display* is discussed in 7. This chapter details how to port the streaming project from the development board to the *arDent* board, and the hardware constraint that prevented it.

Chapter 8 contains the conclusion and next possible developments for this project. Finally, an annex about the internship at *Get Your Way*, and an analysis about their target markets, is located in chapter 9.

Chapter 2

State of the Art

2.1 Physical Interfaces

The goal of this section is to review the existing connectors for video transfer used by cameras, as well as the protocols used in combination.

2.1.1 DVI

The Digital Visual Interface is a royalty-free legacy standard. Three types exist: DVI-A which carries an analog signal, DVI-D which carries a digital signal and DVI-I which can carry both [4]. DVI-A and I are backwards compatible with the VGA standard. Data is transmitted without packetization, using Transition Minimized Differential Signaling (TMDS) over twisted pairs, which ensures good ElectroMagnetic Interference (EMI) resistance. A dual link DVI-D cable can reach a bitrate of 7.92 Gb/s.

2.1.2 HDMI

The High-Definition Multimedia Interface is a licensed proprietary interface managed by the HDMI Forum. HDMI started as an improved version of DVI, also using TMDS over twisted pairs, and is backwards compatible with it with the help of adapters. The latest version of the protocol (2.1b) ditches TMDS for Fixed Rate Link (FRL) and support a maximum bitrate of 48Gb/s [5], as sell as the recently added power supply feature. It usually transmits uncompressed video, but can also use the DSC scheme (see section 2.2.1).

2.1.3 FireWire

Firewire is a licensed legacy serial full-duplex connector, standardized as IEEE 1394 [6]. Maximum bitrate is 3200Mb/s, and power can be transferred as well. Contrary to USB, it uses a peer-to-peer architecture. It can support the GenICam GenTL standard, making it plug and play [7].

2.1.4 PCIe

The Peripheral Component Interconnect Express (PCIe) is a dual-simplex serial bus designed to connect hardware devices inside of computers [8]. It uses a packetized protocol and differential

signaling to transfer data. The latest version of the x16 lanes cables, 6.0, can reach a bitrate of 256Gb/s [9]. PCIe can be used over optic fiber for long range applications. It can support the GenICam GenTL standard, making it plug and play [7]. It can be noted that the PCIe standard requires PCI-SIG membership for commercial use.

2.1.5 MIPI CSI

The Mobile Industry Processor Interface (MIPI) alliance manages different hardware interface standards targeted for mobile applications. The Camera Serial Interface (CSI) is an open licensed standard that notably defines a packetized video transfer protocol. It offers error recovery mechanisms, low power consumption and latency, and multiplexed virtual channels [10]. The use of Differential Pulse Code Modulation (DPCM) allows for lossless compression of data [11]. It is widely used for embedded vision applications. The CSI standard also describes a protocol for camera configuration, the Camera Control Interface (CCI), based on the I^2C protocol. The MIPI Camera Command Set (CCS) [12] and the GenICam GenTL [7] standards allow CSI cameras to be plug and play.
CSI-2 is the most widespread version used, although CSI-3 is more recent. 3 different MIPI physical layer interfaces can be used with CSI-2. Those connectors are smaller than USB, and can deliver data directly to the processor, ensuring low latency [13].

D-PHY A D-PHY lane uses differential signaling on two conductors to transmit data [14]. This allows good EMI resistance. Four lanes can be used at the same time, which combined with the two clock pins result in ten pins. D-PHY can reach a maximum of 36Gb/s with the 3.5 version [15]. D-PHY interfaces can switch between a high speed mode, using differential signaling, and a low power mode, using single ended signaling. The alternate power mode is used during the transition between the two previous modes. D-PHY was designed to carry data over short distances, up to 4 meters, but the closer the less errors and attenuation and thus the more bitrate.

C-PHY C-PHY is more recent than D-PHY. It uses one more wire per lane than D-PHY but the clock is embedded with the useful signal [14]. C-PHY can support 3 lanes of 3 wires at the same time and can reuse the same pins as D-PHY. It is thus fully backwards compatible. C-PHY is designed for the same cable lengths as D-PHY, but offer better EMI resistance and higher bitrate, i.e. 75Gb/s at maximum with the 3.0 version [15]. It also consumes less power.

A-PHY A-PHY was designed to carry data over longer distances, up to 15 meters. It uses an asymmetrical architecture over either a shielded differential pair, a coaxial cable or a Star Quad Cable (SQC) [16]. With a dual lane architecture over SQC, it can reach a maximum bitrate of 64Gb/s in the downlink and 1.6Gb/s in the uplink [15]. It also offers good EMI resistance, and can supply power.

2.1.6 DVP

The Digital Video Port is a parallel interface, aimed at mobile applications like CSI-2. It is simpler to implement, cheaper and more widespread [17]. However, due to its parallel architecture it requires more pins, is more prone to EMI, consumes more power and has a lower bandwidth.

2.1.7 SDI

The Serial Digital Interface is a physical layer standard managed by the SMPTE used to transfer uncompressed video [18]. It is generally used over a coaxial cable but can also be used over optical fiber, offering thus good EMI resistance and long cable length without signal attenuation. Furthermore, it is plug and play, offers low latency and error recovery [19]. However no power or control signals can be transmitted. The latest version, 24G-SDI, reaches a maximum bitrate of 24Gb/s.

2.1.8 Camera Link

The Camera Link is an open, licensed standard maintained by the Automated Imaging Association (AIA). The interface is based on the Channel Link technology and thus also uses Low Voltage Differential Signaling (LVDS) over twin-axial cables [20]. LVDS allows goods EMI resistance and low power consumption. The latest version of the standard, Camera Link High Speed (CLHS), offers 25Gb/s maximum bitrate per cable, long length wires using copper twisted pairs or optical fiber, low latency and single-bit error recovery [21]. It supports the GenICam GenTL standard, making it plug and play [22]. It can be noted that CL connectors are quite large.

2.1.9 CoaXPress

CoaXPress (CXP) is an open, royalty-free standard managed by the Japan Industrial Imaging Association, developed as a successor to Camera Link. It uses a coaxial cable as an asymmetric serial link to transmit data and power [23]. Control messages can thus also be exchanged on the link. The maximum bitrate per cable is 12.5Gb/s in the downlink and 42 Mb/s in the uplink with the latest CXP 2.1 version. Cables can reach 100 meters in length but at a lower bitrate. CoaXPress can also be used over optical fiber. Latency is low and fixed, and the interface is hot-pluggable. It supports the GenICam GenTL standard, making it plug and play [22].

2.1.10 GMSL

The Gigabit Multimedia Serial Link (GMSL) is a proprietary standard that uses a coaxial cable for shielding from EMI, or a twisted pair for a lower cost alternative [24]. It uses a Serializer/Deserializer (SerDes) block: Data is serialized, transmitted over an asymmetric link then deserialized. GMSL 3 can transmit 12 Gb/s in the forward channel and 187 Mb/s in the reverse channel. GMSL is used to extend the range of other physical interfaces, such as HDMI, CSI-2, LVDS or eDP. Cables can reach 15 meters without significant signal fading.

2.1.11 FPD Link

Flat Panel Display Link is an open, royalty-free standard. It uses differential signaling and a SerDes block to transmit data and power [25] over either a twisted pair, a coaxial cable or a star-quad cable [26]. Cables can thus reach 15 meters without signal fading, and power consumption is low. Control signals can also be exchanged on the same link, using the I^2C or SPI protocols. FPD Link is used to extend the range of other physical interfaces, such as HDMI, CSI-2 or LVDS. The latest version, FPD link IV, reaches a bitrate of 13Gb/s per cable.

2.1.12 Ethernet

Ethernet cables use the Internet protocol suite to transfer data. Length can reach 100 meters and bitrate 40Gb/s for twisted pairs cables [27], while optical cables can reach several kilometers and 400 Gb/s [28]. Cables that comply to the Power over Ethernet (PoE) standard can transfer also power in addition of data. Ethernet can support the GenICam GenTL standard, making it plug and play [7]. Multiple application layer protocols can be used for video streaming, such as: RTSP, RTMP, DASH, HLS, WebRTC [29]. However, in industrial settings, GigE vision is preferred.

GigE The Gigabit Ethernet standard is a licensed standard which mainly describes two application layers protocols : the GigE Vision Control Protocol (GVCP) and the GigE Vision Stream Protocol (GVSP) [30]. It was designed for uncompressed video transfer, but since version 2.0 the following compression formats are also supported: JPEG, JPEG 2000 and H.264 [31]. The latest version, GigE 2.2 supports a bitrate of 10Gb/s. It can be noted that the CPU usage is slightly higher with GigE interfaces than with FireWire, USB 3.0, CameraLink or CoaXPress [32]. GigE compliance implies GenICam GenTL support [22].

2.1.13 Thunderbolt

Thunderbolt is a proprietary licensed standard. It defines a packetized protocol that combines DisplayPort signals for video transfer and PCIe signals (see section 2.1.4) for data transfer on the same link [33]. It uses the mini DisplayPort connector, and either optical or electrical cables. With the former it can also carry power. It can reach a bitrate of 10Gb/s with version 1, and 20Gb/s with version 2.
Thunderbolt 3 and later versions use the USB-C connector, but are not used by cameras and will thus not be discussed.

2.1.14 USB

The Universal Serial Bus (USB) is a standard managed by the USB Implementers Forum (USB-IF) which defines connectors for data and power transfer, as well as the protocols to use them. USB cables are either electrical, or optical for longer range and better EMI resistance.

2.1.14.1 USB Protocols

Multiple USB standards exist, with different bitrates, but they are not available on every connector. USB protocols are more complex than others like CSI-2, but they make up for it with their widespread support and by being plug-and-play [34]. The original USB standard was created to provide a bi-directional, low cost, plug-and-play, low to mid speed bus. It was intended to be used to connect peripherals and phones to PCs. This explains why the first USB protocols implement a master-slave architecture.

USB 2.0 The USB 2.0 protocol is a packetized half-duplex protocol. It uses a master-slaves architecture, named host and devices [1] in the standard [35]. The USB host initiates the communications

[1] A device which provide a capability can also be called a function

by polling the device and supplies them with power. There can only be one in the system and the role is fixed.
The USB 2.0 protocol allows a maximum bitrate of 480Mbit/s, but it only reaches around 240-320Mbits/s because of hardware interface limitations and driver overheads [36].

USB 3.0 USB 3.0 is the successor to USB 2.0. USB 3.0 cables have 5 additional wires than 2.0 ones, two twisted pairs and a ground wire, which form the SuperSpeed bus. It allows USB 3.0 to be dual simplex while also retaining the USB 2.0 channels, thus keeping backwards compatibility [37]. The USB 3.0 protocol is an improved version of the USB 2.0 protocol, with better flow control and error correction for instance. This allows USB 3.0 cables to reach higher bitrate, i.e. a maximum theoretical bitrate of 5Gb/s but an effective one of 3.2Gb/ [38], and a higher power supply. Furthermore, hosts do no need to poll before every data exchange, lowering the computational load. The connector can be differentiated from the 2.0 ones thanks to their blue color.

USB 3.1 USB 3.1 is the successor to USB 3.0. It uses the same cable architecture, but the clock used on the bus is twice as fast and data encoding is more efficient [39]. This faster bus is named SuperSpeedplus. Bitrate is thus twice as high as USB 3.0, i.e. a maximum theoretical bitrate of 10Gb/s but an effective one of 7.2Gb/s [38].
USB 3.0 and 3.1 were renamed to USB 3.1 gen 1 and gen 2 afterwards.

USB 3.2 USB 3.2 is the successor to USB 3.1. It uses two SuperSpeedplus buses, which results in double the bitrate compared to USB 3.1, i.e. a maximum theoretical bitrate of 20Gb/s but an effective one of 16Gb/s [38]. Contrary to the previous protocols, it is only available on USB-C.
USB 3.0, 3.1 and 3.2 were renamed to USB 3.2 gen 1x1, gen 2x1 and gen 2x2 afterwards. SuperSpeed USB 5Gbps, SuperSpeed USB 10Gbps and SuperSpeed USB 20Gbps are also used as commercial names.

USB4 USB4 2.0, or gen 4x2, is the latest version of the USB standard. It can reach a bitrate of 80Gb/s using the same dual lane architecture as USB 3.2 [40]. It is however not used yet by cameras and will thus not be discussed further.

2.1.14.2 USB Classes

When a USB device gets connected to a host, it gives information about its capabilities in a process called configuration. One given attribute is the USB class of the device. USB classes are groups of devices that share common commands and descriptors [41]. The UVC and USB3 Vision classes in particular are widely used to connect cameras.

USB Video Class The UVC regroups devices that necessitate or perform video transfers [42]. Devices that implement the UVC standard can transfer video with the following video coding formats:

- Uncompressed video
- MPEG2-PS
- MPEG1-SS
- MPEG-2 TS

- MPEG-4 SL
- MJPEG
- H.264
- VP8
- VC1
- DV

There is also the possibility to use vendor-defined formats, such as:

- H.265

It can support the GenICam GenTL standard [7].

USB3 Vision Similarly to GigE, USB3 Vision is a licensed standard that is designed for industrial settings. It redefines some parts of the USB standard, such as device identification, control and how data is exchanged, i.e. via the USB3 Vision Streaming Protocol (UVSP) [43]. USB3 Vision implies GenICam GenTL compliance [22].

2.1.14.3 USB Connectors

USB-A USB-A receptacles are only for hosts.

USB-B USB-B receptacles are only for devices.

Micro USB Micro USB-AB and micro USB-B receptacles can be used for both hosts and devices. They have one more pin than USB-A and USB-B receptacles, the ID pin [44]. It is used used to specify if a host or a device is connected.
Devices whose only USB receptacle is a Micro USB-AB one, and can be used both as host or device, are called On-The-Go (OTG) devices [45]. The host negotiation is done with the Host Negotiation Protocol (HNP) for USB 2.0, and the Role Swap Protocol (RSP) for USB 3.0. Some devices are not able to negotiate their role and instead rely on the plug used to determine it. These are called Dual Role Devices.

USB-C USB-C was designed to be smaller, more robust and easier to use than previous USB connectors. Indeed, it can be plugged upside-up and upside-down, and can be used for both hosts and devices. Host and devices are called Downstream Facing Port (DFP) and Upstream Facing Port (UFP) in the USB-C nomenclature, and the role of a port can be negotiated. The standard specifies that unshielded twisted pairs, shielded twisted pairs as well as coaxial and twin-axial wires can all be used inside a USB-C cable for the signal pairs. Simple wires are used for sideband signaling and power supply [46]. Different kinds of cables are defined in the standard:

- USB Full-Featured Type-C cables, i.e. USB-C cables that support USB 2.0, USB 3.2 and USB4 data operation as well as Power Delivery.
- USB 2.0 Type-C cables with a USB 2.0 Type-C plug at both ends
- Captive cables, i.e. cables that has one USB connector and is either permanently attached or has a non-USB connector. In this case the USB connector is either a USB Full-Featured Type-C plug or USB 2.0 Type-C plug.

- Active cables, i.e. cables with embedded signal conditioning circuits. They can be longer without signal fading.

Power Delivery is a USB-C protocol that provides flexible power supply [47]. Contrary to previous USB power protocols, the power flow direction is negotiated and does not depend on the roles of the devices. Furthermore, the maximum power rate is higher. Power Delivery packets are sent on the Configuration Channel (CC). This channel can notably be used as a side-band channel, e.g. for discovery of cable capabilities, Data Role Swap (either DFP or UFP) and alternate modes negotiation.
Alternate modes, or alt modes, are protocols defined by standards or vendors, that utilize the reconfigurable pins of the USB-C cable to supersede the USB protocol. Popular alt modes include:

- **HDMI**
  See section 2.1.2
- **DisplayPort**
  Protocol designed for streaming video to a display.
- **MHL**
  Adaptation of HDMI targeted for mobile devices.
- **Thunderbolt** (3.0 onward)
  See section 2.1.13

2.2 Video Compression Formats

The goal of this section is to review the existing video compression formats used by cameras.

Compression designates the suppression of information in some data, in order to lower its size. It can be lossless, if the discarded information was redundant, or lossy otherwise.
In the case of video compression, four type of redundancy can be identified:

- **Spatial redundancy**
  In any non random array of pixels, e.g. a picture, pixels values are correlated with their neighbors.
- **Temporal redundancy**
  In any non random series of array of pixels, e.g. a video, pixels values are correlated with values from neighboring arrays.
- **Entropy redundancy**
  In any non random signal, some code values appear more frequently than others and hold thus less information. Representing those frequent values by short codes will reduce the number of bits to transmit.
- **Perceptual redundancy**
  Some information is not visible to the human eye and is thus irrelevant in applications targeted for humans. For instance, high spatial resolution and high sampling rates are not discernible to the human eye, especially for the chrominance. Compression algorithms discarding this kind of information are said to be visually lossless.

2.2.1 DSC

Display Stream Compression is a visually lossless compression scheme designed for real-time video streaming from a source to a display [48]. It is thus computationally light to ensure low latency. DSC can be notably used over the HDMI protocol.

2.2.2 MJPEG

Motion JPEG (MJPEG) utilizes the JPEG scheme to separately compress each frame, without using temporal redundancy. It is thus computationally lighter and insensible to motion complexity, but less efficient at compression than algorithms that do use it [49]. Since frames are all intra-coded, any frame can easily be extracted as a standalone image and errors in a frame do not affect others, which is useful in case of transmission errors. Thanks to the low compression, it reaches higher quality than other more efficient schemes. M-JPEG was never standardized, so multiple implementations coexist, such as the UVC one. Indeed, although frame compression uses the JPEG standard, organization of the stream of frames depends on the MJPEG implementation.

2.2.3 MJPEG 2000

Similarly to MJPEG, MJPEG 2000 separately compress each frame but uses the JPEG 2000 scheme. This format thus offers slightly more compression than MJPEG but at the cost of more complexity [50]. It was standardized by the International Organization for Standardization (ISO).

2.2.4 MPEG 4

The Moving Picture Experts Group (MPEG) is a working group that creates standards about coded representation of digital audio, video, 3D Graphics and genomic data [51]. The MPEG video coding formats are open, licensed standards.

Part 2 of the MPEG 4 standard covers video compression. MPEG 4 supports low bitrate applications, e.g. mobile units or streaming over the internet, as well as high bitrate high quality applications, e.g. playing a video from local storage [50].

The MPEG 4 standard supports multiple applications, using different sets of algorithms, known as profiles [52]. In order to simplify decoders, not all profiles have to be implemented. Furthermore, it is also allowed by the standard to only support a set of parameters values, known as levels. Profiles and levels are also present in other standards like H.264 or H.265.

2.2.5 H.264

The part 10 of the MPEG 4 standard also covers video compression. It is better known as H.264, or Advanced Video Coding (AVC). It was designed to provide better compression for a fixed quality compared to existing formats, without increasing the complexity too much. For instance, H.264 compress 80% more than MJPEG, and 50% more than MPEG 4 part 2 [53]. It was also meant to be flexible and used for many applications such as Internet streaming, TV broadcast, storage on Blu-rays,... Indeed, it offers good error robustness to accommodate transmissions errors, a straightforward syntax to simplify implementations, and exact match decoding, i.e. it defines how calculations are made by the encoder and decoder in order to avoid accumulating errors. Furthermore, it can handle low and high bitrates, resolutions or latencies. It is widely supported nowadays.

2.2.6 H.265

H.265, also known as High Efficiency Video Coding (HEVC),is the successor of the H.264 format, designed to offer more compression for similar quality. Indeed, it can reach 50% reductions in bitrate without quality loss compared to H.264 [49]. However, it is more complex and not as widely supported as H.264.

2.3 Case Study

This section contains a comparison of the different physical interfaces described in section 2.1, with the intent of finding the most relevant one for a possible new version of the *arDent Display*. The analysis is not done for the current *arDent* board since only one connector is present on it: a USB-C port.

2.3.1 Scope Statement

Here follows a description of the needs of Get Your Way concerning the physical support used to connect the camera.

Limiting requirements:

- Reasonable computational load, to keep power consumption low
- Wide support by cameras, for ease-of-use by customers
- Plug and play, for ease-of-use by customers
- Power supply through cable, to allow light cameras without battery to be used
- Control through cable, to allow interactivity with the camera, e.g. to take a photo at a specific time
- Bandwidth sufficient for a stream of 854x640-pixel frames at 30 FPS
- Small connector receptacle, since the *arDent Display* is relatively small itself
- Royalty-free, to reduce expenditures

Non limiting requirements:

- Cable length up to 0.5m
- Low EMI resistance

These two non limiting requirements allow the use of electrical cables. Optical cables will thus not be considered as they are more expensive and consume more power [54] .

2.3.2 Technology Comparison

As a first sorting, the following standards can be ditched:

- SerDes standards, as cable length and EMI resistance are non limiting requirements:
  FPD Link, GMSL
- Licensed standards, to limit expenditures:
  HDMI, PCIe, CSI, Camera Link, Thunderbolt

- Obsolete legacy standards:
  DVI, FireWire
- Parallel interfaces, as they are less energy efficient than serial interfaces:
  DVP

4 standards remain: SDI, CoaXPress, Ethernet and USB. Table 2.1 compares them using relevant characteristics. USB2 and USB3 are both assumed to be used with a USB-C receptacle, Ethernet with a RJ45, SDI and CoaXPress with a micro-BNC.

	24G-SDI	**CoaXPress 2.1**	**Ethernet Cat 8.2**	**USB2**	**USB3.2**
Maximum bitrate [Gb/s]	24	12.5	40	0.48	20
Computational load [32]	low	low	medium to low	medium	low
Plug and play	yes	yes	yes	yes	yes
Power through cable	no	yes	yes	yes	yes
Control through cable	no	yes	yes	yes	yes
Receptacle size [mm]	7.48x7.48	7.48x7.48	16x13.24	8.34x3.26	8.34x3.26
Compression	none [2]	none	possible	possible	possible

Table 2.1: Physical interfaces comparison.

2.3.3 Summary

SDI can not be used because of the lack of power supply and control through cable. Since Ethernet and USB2 have higher computational load than CoaXPress and USB3, they are less suited for embedded applications and should also be avoided. CoaXPress is targeted for industrial uses, and cameras are thus rare and expensive.

The USB-C connector is thus the most suited physical interface for this application. The protocol used in combination should be at least 3.0 to benefit from the low computational load, or a more recent version like 3.2.

[2] JPEG support was added in 2024, but the intended and most widespread use of SDI is uncompressed video transmission [55]

Chapter 3

USB 2.0 Standard

This chapter describes the vocabulary and concepts from the USB 2.0 protocol [35] [56] that are necessary to understand the description of the USB Host Driver in section 6.4, with a focus on high-speed devices.

3.1 General Architecture

As mentioned in section 2.1.14.1, the USB 2.0 protocol uses a master-slave architecture. The master is called a host, whereas slaves are called devices. Devices that provide a capability to the host are called functions. The host is expected to be implemented on a powerful device such as a computer, and most of the complexity of the USB protocol is thus managed by the host. For instance, it is the host that supplies power to the devices, and only hosts can initiate USB requests to connected devices. Indeed, devices can not send USB packets unprompted, they must first be polled. All this allows to simplify USB peripherals. USB hosts are supposed to support every USB class of devices (see section 3.2). Simpler hosts were later added to the USB specification : so-called targeted hosts, which only support only a specific set of classes. Some devices can either be used as host or function, and are called OTG devices.

A USB system thus consists of one host and at least one device. The host initiates every transfer, and can target them to the right device by using their address, which the host assigns. Multiple devices can thus be in operation at the same time. Hubs are special USB devices that allow to use one USB port to connect multiple devices, including other hubs potentially. A USB system is thus a logical tree with the host at the root and devices at the leaves, with hubs in between. Figure 3.1 represents the bus tiered star topology. Only 7 tiers are allowed, and only 5 non-root hubs can be chained, in order to keep hub and propagation delays reasonable. A compound device is represented on the diagram: it is a device that contains an embedded hub.

A USB host is made up of two components. The first is the host controller, implemented both through software and hardware. It implements the following features:

- Detection of device attachment and removal
- Management of data and control flows between the host and devices
- Management of power supplied to the devices
- Management of bus activity

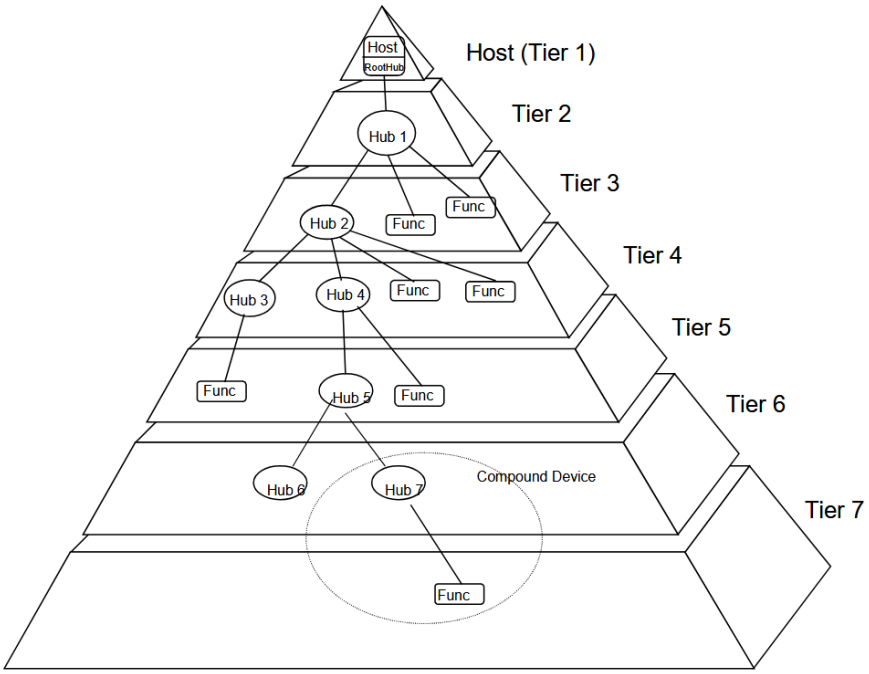

Figure 3.1: USB bus topology
Source: [56]

The second component is the root hub. It is a USB hub embedded in the host that connects the host controller to every USB port of the device. The root hub is handled by the hub class driver like any other hub, but the interface between this hub and the host controller depends on the hardware implementation. Indeed, USB transactions may not be necessary to exchange data between the controller and the root hub, given the close integration possible between the two.

3.2 USB Devices

USB devices are classified into groups that share common commands and descriptors, called classes. They are supported by specific class drivers. Many classes are defined by the USB standards and custom ones are also allowed. A descriptor is a data structure that contains information about the device capabilities. Descriptors are either part of the USB 2.0 standard itself, standardized in class-specific standards or vendor-defined. Each descriptor contains at least a length field for parsing purposes and a descriptor type field. Values stored on multiple bytes use the little endian encoding scheme.

The main descriptor is the device descriptor: it is a unique and contains information about the device itself, such as the implemented USB specification number, product ID, USB class and subclass [1], indexes to strings describing the device, and numbers of configurations.

Configuration defines which interfaces can be used and what is the power source of the device: self powered or bus powered, as well as the max drawable current. Only one configuration can be enabled at a time.

[1] Classes and subclasses can also be defined separately for each interface

Interfaces are the layer below configurations: they represent a particular function implemented by the device. Interface descriptors contain USB class and subclass fields, and serve as anchor for endpoint descriptors as well as class-specific descriptors. Devices with interfaces that implement different functions are called composite devices. The host can switch interface during device operation in order to access other functions. For instance, a camera with a microphone which possess a video and an audio interface. An interface may also have alternate settings, i.e. interfaces with the same interface number but with different endpoints descriptors. As only one alternate setting can be active at a time, the same endpoint numbers can be reused.

Figure 3.2 is a diagram that describes a composite device with multiple configurations, interfaces, and alternate settings.

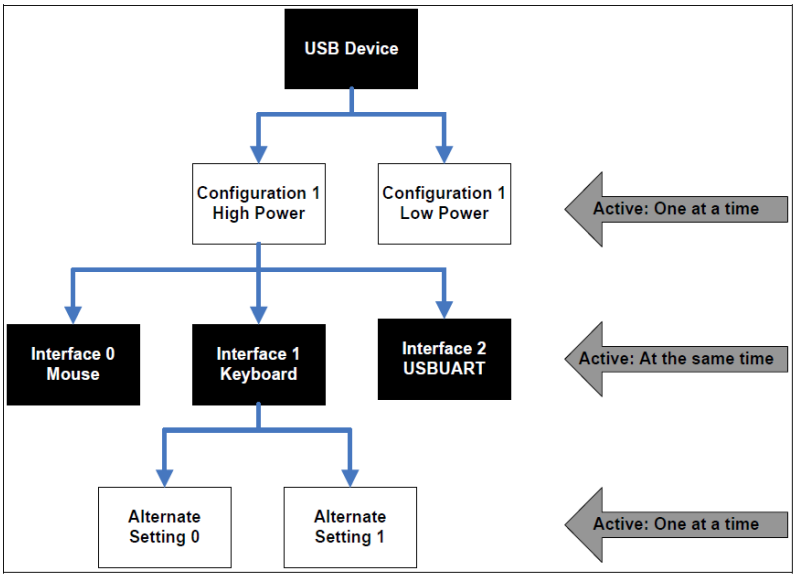

Figure 3.2: USB device configuration
Source: [35]

3.3 USB Communications

3.3.1 Pipes

USB communications between the host and a given device are done through a pipe. A pipe is a connection pathway between the host controller and a specific addressable buffer of a device, called an endpoint. An endpoint is either used to transfer data to the host (IN type) or to transfer data the device (OUT type). Multiple pipes to the same device can be set, as long as they use a different endpoint, as depicted on figure 3.3. In this example, 3 pipes are set : the default control pipe, a IN data pipe and a OUT data pipe. The default control pipe is a special bi-directional pipe which always uses endpoint 0. It exists on every USB device as long as they are powered, and is used to access device configuration, status and control information.

They are two types of pipes : message and stream pipes. The data exchanged with a message pipe transaction follows a packet structure defined by the USB 2.0 standard, while no structure is

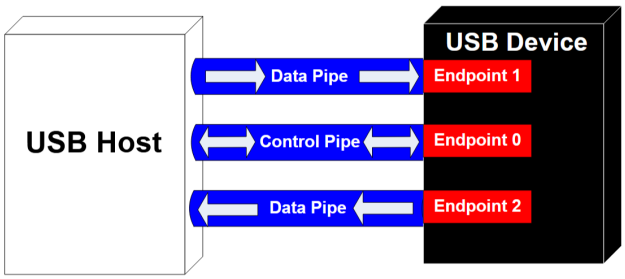

Figure 3.3: USB pipe model
Source: [35]

imposed for stream pipes. The only message pipes are the control pipes, described in section 3.3.3.1. As mentioned earlier, a pipe is defined by 2 variables : a device address and an endpoint number. However, it is not enough information to execute a transfer, and the host has to store additional pipe-related data, such as : bus access frequency, transfer type, direction and the Maximum Packet Size (MPS), i.e the maximum transaction length. Pipes other than the default control pipe are set during the configuration process, described in section 3.4.3.

3.3.2 Transfers

USB transfers are defined by the USB specification as follows : "one or more bus transactions to move information between a software client and and its function". There are multiple types of transfers defined in the specification, described in the following sections. They can be separated into two categories : periodic transfers, which offer bandwidth and latency guarantees, and asynchronous transfers, which implement a best effort model. Isochronous and interrupt transfers are periodic, whereas bulk and control transfers are asynchronous. All transfers except isochronous ones implement a transaction retry mechanism upon error detection.

In order to implement the periodic transfers, the USB specification defines a $1ms$ time base called a frame, and a $125\mu s$ time base called microframe. Frames are the base used by low- and full-speed devices, whereas the microframe is used by high-speed devices. Each transfer type describes what transactions and how many can be executed during a (micro) frame [2]. Transfers denoted as periodic are given an opportunity to be executed every N number of (micro) frames, with N the polling period, which is endpoint-dependent.

Figure 3.4 describes the USB communication protocol from a time perspective. Each (micro) frame starts with a Start Of Frame (SOF) transaction, to mark the beginning of a (micro) frame, and synchronize host and device clocks. Multiple transactions are then executed, up to 3. Transactions are done by sending 3 packets: a token packet containing the address, endpoint number and direction, a data packet containing the data requested by the host, and a handshake packet [3] to acknowledge the transaction.

The handshake packet can also be used by a device to communicate a STALL condition to the host. Two situations warrant this:

- Functional STALL: the function is unable to transmit or receive data over a specific pipe. The

[2](micro) frame means frame or microframe, according to the USB speed.
[3]Except for isochronous transactions

host has to explicitly clear this STALL. The host can never return a STALL during handshake but can set a halt bit on a device endpoint, which triggers a STALL on that endpoint. This commanded STALL can be used for instance if the host wants to stop receiving data from an IN endpoint temporarily.

- Protocol STALL: the function is unable to complete the requested control transfer, for instance because the data content of the transfer is invalid. This STALL is automatically cleared upon the next control transfer.

A control pipe may support functional STALLs, but this is not recommended by the standard. The standard also states that devices do not need to return STALL for class-specific and vendor-specific requests.

The USB protocol, detailing the execution of bus transactions, will not be described any further as it is implemented by the USB host hardware module, whereas it is the controller software component that is of interest here.

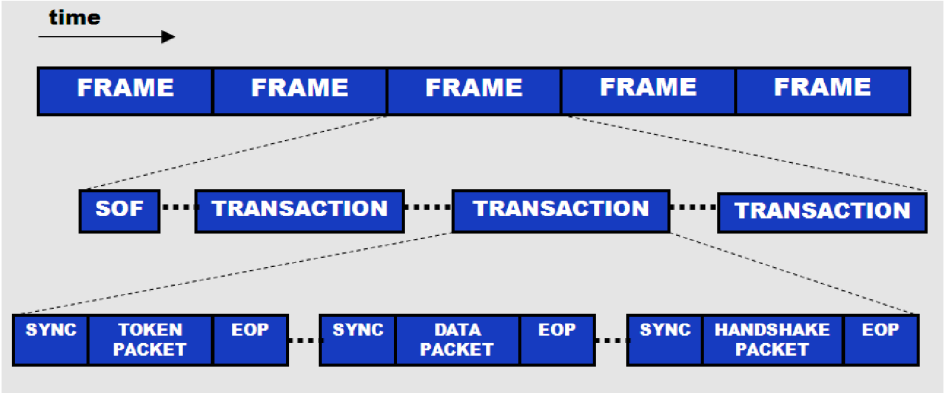

Figure 3.4: USB communications
Source: [35]

3.3.3 Asynchronous Transfers

3.3.3.1 Control transfers

Control transfers are used to send commands, configure the device and obtain status information. Some commands are supported by all USB devices, while others are only implemented by specific classes. Control transfers are done using a control pipe, which is a message pipe. They have priority over bulk transfers. As mentioned earlier, the default control pipe is always present, so that the device can be configured with it. The standard also allows a device to provide other control endpoints in addition to endpoint 0, in order to set up other control pipes.

A control transfer consists of 3 steps. First, the host performs a set-up transaction to specify the request. Afterwards, data transactions are performed in the direction specified during the set-up transaction. Finally, the device returns status information to the host by performing a status transaction.

The different fields of the set-up transaction packet are specified by the standard, but the interpretation of the values they hold is class-dependent. Table 3.1 details those fields. The data packet content depends entirely on the USB class.

Offset	Field	Size (B)	Value	Description
0	**bmRequestType**	1	Bitmap	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	Value	Specific request
2	**wValue**	2	Value	Word-sized field that varies according to request
4	**wIndex**	2	Index or Offset	Word-sized field that varies according to request; typically used to pass an index or offset
6	**wLength**	2	Count	Number of bytes to transfer if there is a data stage

Table 3.1: Format of Set-up data packet
Source: [56]

3.3.3.2 Bulk transfers

Bulk transfers are used to transfer large amounts of data. They utilize all available bandwidth but do not offer any guarantee on speed or latency. The highest MPS allowed is 512 bytes. Although the theoretical maximal USB 2.0 bandwidth is announced to be $60MB/s$, the specification actually mentions that is not reachable. Indeed, the bus frequency and microframe timing limit the number of successful bulk transactions within one microframe, $N_{t/\mu f}$, to 13. The maximum amount of periodic data $D_{\mu f}$ transferable during one microframe is thus:

$$max(D_{\mu f}) = max(N_{t/\mu f}) * max(MPS)) \tag{3.1}$$
$$= 13 * 512 = 6656B/\mu frame \tag{3.2}$$

The maximal bulk bandwidth B_B is thus :

$$max(B_B) = N_{\mu f/s} * max(D_{\mu f}) \tag{3.3}$$
$$= 8000 * 6656 = 53.248MB/s \tag{3.4}$$

With $N_{\mu f/s}$ the number of micro-frames in a second.

3.3.4 Periodic Transfers

Periodic transfers provide a bounded latency and can reserve bandwidth. The entire bandwidth can not be reserved however : with a high-speed device for instance, 80% of a microframe can be used for periodic transfers at most. This allows to save some bandwidths for asynchronous transfers.

The USB 2.0 specification states that for a high-speed device, the MPS is limited to 1024 bytes and the number of transactions per micro-frame $N_{t/\mu f}$ is limited to 3. The maximum amount of periodic data transferable during one microframe $D_{\mu f}$ is thus:

$$max(D_{\mu f}) = max(N_{t/\mu f}) * max(\text{MPS}) \tag{3.5}$$
$$= 3 * 1024 = 3072B/\mu frame \tag{3.6}$$

The maximum amount of data that can be transferred during an entire frame is thus :

$$max(D_f) = N_{\mu f/f} * max(D_{\mu f}) \tag{3.7}$$
$$= 8 * 3072 = 24576B/\text{frame} \tag{3.8}$$

With $N_{\mu f/f}$ the number of micro-frames within one frame . Since a frame is equal to $1ms$, the maximum bandwidth reservable for periodic transfers B_p is :

$$max(B_p) = max(D_f) = 24576B/ms \tag{3.9}$$
$$= 24.576MB/s \tag{3.10}$$

3.3.4.1 Isochronous transfers

Isochronous transfers require a guaranteed latency and share of the USB bandwidth. They use stream pipes, i.e. the content structure of the transactions depend on the USB class used. The host is allowed by the specification to poll at a higher rate than the one advertised, and devices must be able to handle it. In the case where data is no yet ready to be written into the IN endpoint, the device has to send a 0-length packet.

3.3.4.2 Interrupt transfers

Interrupt transfers offer both reliable delivery of data and bounded latency. They are used when small amounts of data have to be transferred infrequently. Contrary to what the name might suggest, the host has to poll the device at a defined frequency, and the device has to acknowledge all transactions until it has data to send. Reception of said data by the host concludes the interrupt transfer.

3.4 Device Attachment

This section contains a simplified description of how a device gets configured for use upon attachment. There are 3 main processes : first the device gets detected during dynamic detection. Then is it given an address during enumeration. Finally, the host learns about the device capabilities and configures the device during configuration.

3.4.1 Dynamic detection

As described earlier, a USB device can only be connected through a hub, which can either be the root hub or a hub device. Upon attachment, the device raises the voltage of one of the D lines of the port, depending on the device USB speed. As can be seen on figure 3.5, it is done using pull-up

resistors. Depending on the D line used, the hub can learn if the device supports low- or full-speed. High-speed support can not be inferred at this stage. The device can only draw $100mA$ from the bus for the time being.

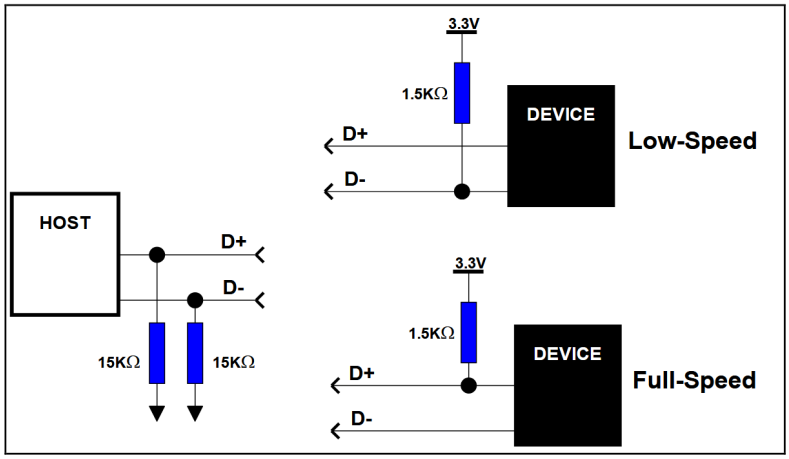

Figure 3.5: USB device detection
Source: [35]

3.4.2 Enumeration

The host has to query the hub in order to learn about the attachment, and the device speed. It uses an interrupt pipe for that purpose. The host then has to send a reset the device, by commanding the hub to do so. During this reset, a high-speed device can use the D lines to send a specific binary sequence which inform the hub about its high-speed support. Once the device has left the reset state, it can now respond to control requests. The default control pipe associated with this device always uses address 0, which is reserved for devices undergoing enumeration. This implies that only one device can be enumerated at a time.

The host then sends a GET_DESCRIPTOR request to the device, to obtain the device descriptor. Only the eighth byte, containing the MPS of endpoint 0, is of interest. Devices are thus allowed to only send the first eight descriptor bytes. A device reset is executed again, so that the second GET_DESCRIPTOR does not perturb the device. Finally, the hosts sets an address for the device. All following communications now utilize the new address, until the device is detached, reset, or the host reboots.

3.4.3 Configuration

The host begins by asking for the configuration descriptor. It uses this descriptor to learn about the device capabilities, in order to choose the correct driver to manage the device. Finally, the host issues a SET_CONFIGURATION request. From that point on, configuration is over and the device can draw the maximal agreed power from the bus. The device can now also respond to class-specific requests, which likely involves other configuration commands before it can be used.

Chapter 4

UVC Standard

As mentioned in section 2.1.14.2, the USB Video Class (UVC) is a USB class which implements video-related functionalities, such as remote control of camera and video streaming over USB. This section describes the UVC standard concepts needed to understand the UVC driver implementation in section 6.8. The focus is thus on a USB host which receives a video stream through.

Every UVC version is backwards compatible: either the host or the peripheral can implement an outdated version and still remain compatible. Version 1.0 is thus sufficient to implement basic video streaming while still being supported by a wide array of devices, which is why it is the version described in this section. As the 1.0 specification is no longer available online, version 1.1 was used as source [57]. The revision history details the changes made by version 1.1, which only consists of optional functionalities and a few clarifications.

4.1 UVC Operational Architecture

Video functions are implemented at the interface level, via two UVC-specific interface subclasses. A UVC device contains one VideoControl (VC) interface and may contain multiple VideoStreaming (VS) interfaces. Different settings, stored in logical objects named controls, can be modified by the host. Controls belong to different addressable entities , i.e. : either a terminal, unit, interface or endpoint. It can be noted that the term entity just designates terminals and units however.

4.1.1 VideoControl interface

The VC interface is used to configure the video device itself, e.g. the camera. VC controls are located inside units and terminals and can be used to modify the zoom or focus for instance. The VC interface is thus unique, and no alternate setting can be defined. It contains two endpoints :

- A control endpoint, implemented using the default endpoint (number 0). It is used to access controls of units and terminals. It is also used to access statuses of the video function.
- An optional interrupt endpoint, used to return status information about any addressable entity of the entire video function (VC and VS interfaces). It is located in this interface because it is always present, whereas the VS interface is optional.

Controls of this interface are initialized with default values. This implies that streaming is possible without modifying them.

4.1.2 VideoStreaming interface

VS interfaces are used to configure and execute transfers of video data into or out of the function. VS controls can be used to modify the video format or frame dimensions for instance. Multiple VS interfaces might be exposed to allow concurrent video streams with different characteristics. The use of interface alternate settings allow to change the capabilities of a given endpoint, such as the bus access period or the MPS. The endpoint and alternate choice is done during the negotiation described in section 4.4. The specification also allows to provide no VS interface, which is useful in the case where the peripheral camera is just remotely controlled via USB but does not stream any video data for instance.

A given VS interface can only contain one bulk or isochronous endpoint for video transfer, and one optional bulk endpoint for still image transfer. The associated endpoints may either use IN or OUT as direction, depending on whether the device transmits or receives the video stream.

Some parameters of this interface have to be negotiated between the host and the device in order to set up the video stream. This negotiation is detailled in section 4.4. It is only after this negotiation that a VS interface can be chosen, and the related endpoint initialized.

4.2 UVC Descriptors

As described in section 3.2, a device can have multiple interfaces in the same configuration, which could potentially belong to different independent video functions if it is a composite device. Interfaces that belong to the same video function are regrouped into a video interface collection, described in an interface association descriptor. Subclass-specific descriptors are then used to store additional information about those interfaces, such as the supported video formats.

Figure 4.1 is a tree structure describing the device descriptor of the UVC camera used in chapter 6. Child nodes contain additional information about their parent node. This structure can be deferred from the configuration descriptor sent during the configuration process (see section 3.4.3): descriptors are sent from top to bottom, whereas dependencies between descriptors are described in the specification. For instance, when an interface descriptor is found during parsing of the device descriptor, it implies that every following descriptors are related to it, until another interface descriptor is found.

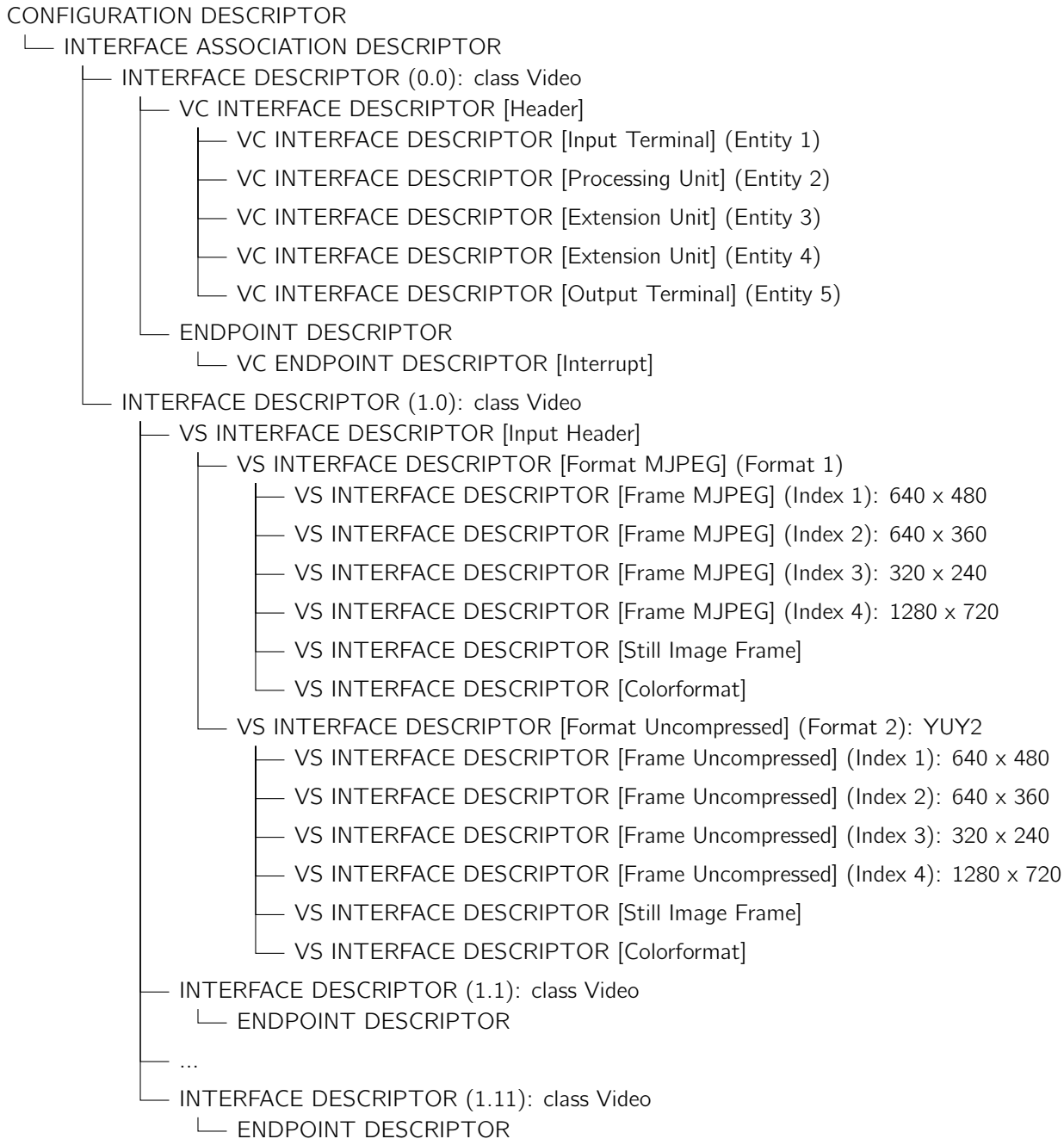

Figure 4.1: UVC test camera configuration descriptor

4.3 UVC-Specific Requests

The UVC standard specifies class-specific requests using the control transfers described in section 3.3.3.1. These UVC control transfers make use of the protocol STALL mechanism described in section 3.3.2 in the following cases: unsupported command, invalid target entity, unexpected data length, or invalid data content. The host can learn about the STALL cause by requesting the value of the VC Request Error Code Control for VC requests.

Since control transfers are used, set-up packets must follow the structure defined in table 3.1. The *bmRequestType* and *wLength* have the same meaning as defined in the table. The interpretation of *wValue* depends on the targeted entity. There are two types of UVC-specific requests, which use different values of *bRequest*. They are described in the following sections. *wIndex* is used to target an addressable entity, which depends on the control that is accessed and the *bRequest* type. An entity (unit or terminal) can be stored in the high byte and an endpoint or interface can be stored in the low byte.

4.3.1 Set request

Requests with *bRequest* set to SET_CUR are used to set an attribute of a specific entity of the video function. For most entities, *wValue* stores the Control Selector (CS) field, which is used to select a specific control in an entity which own several. If the entity only has one control, it can be used to store other data instead. The actual value to set in the control is stored in the data stage of the transfer.

4.3.2 Get request

Multiple requests which are used to retrieve attribute values from specific entities are defined, each using a different *bRequest*. These allow to specify which attribute of a control parameter to request.

- Current setting attribute (GET_CUR)
- Minimum setting attribute (GET_MIN)
- Maximum setting attribute (GET_MAX)
- Default setting attribute (GET_DEF)
- Resolution attribute (GET_RES)
- Data length attribute (GET_LEN)

Additionally, information about the control can be obtained using GET_INFO as *bRequest*.

4.3.3 Summary of VideoStreaming requests

As mentioned in section 4.1, some VS controls have to be modified in order to set up a stream, while VC controls all have default values. It is thus relevant to summarize the VS different requests in a table.

Table 4.1 details the possible requests to the VS interface. The Control Selector (CS) and the interface numbers must be stored in the high byte.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001	SET_CUR	CS	Interface	Length of parameter block	Parameter block
10100001	GET_CUR				
	GET_MIN				
	GET_MAX				
	GET_RES				
	GET_DEF				
	GET_LEN				
	GET_INFO				

Table 4.1: VideoStreaming request fields
Source: [56]

4.4 VS Parameter Negotiation

Contrary to VC parameters, it is not sensible to set default values for some VS parameters related to the video stream format and encoder. These particular parameters include the format, frame size and frame rate of the stream. Key frame rate and compression factor are also included, if relevant to the video format used. A stream can not be started before the host and the device negotiate the values of those special parameters.

Theses negotiable parameters are handled via the Video Probe and Video Commit controls. The Probe control is used for the negotiation itself, whereas as the Commit Control is used to set the parameters to the values previously negotiated. Table 4.2 summarizes the Probe and Commit controls parameters that are relevant to establish a basic video stream with a frame-based format (e.g. MJPEG or uncompressed formats). The offset field is relative to the control data structure, which is the data payload used in GET_CUR and SET_CUR requests. It is 26-byte long is UVC 1.0. Later versions of the UVC standard added fields to these controls, but they are optional to maintain backward compatibility.

Offset	Field	Size (B)	Value	Description
0	**bmHint**	2	Bitmap	Hint bitmap, used by the host to inform the VS interface about which fields shall remain constant during parameter negotiation. The only relevant hint for frame-based formats is **dwFrameInterval**, stored in the first bit.
2	**bFormatIndex**	1	Number	Video format index, corresponding to the order of the VS format descriptors.
3	**bFrameIndex**	1	Number	Video frame index, corresponding to the order of the VS frame descriptors of the chosen format.
4	**dwFrameInterval**	4	Number	Frame interval in 100 ns units. The value must be in the range defined in the frame descriptor.
18	**dwMaxVideoFrameSize**	4	Number	Maximum video frame size in bytes. Relevant if compression is used.
22	**dwMaxPayloadTransferSize**	4	Number	Maximum number of bytes that the device can transmit or receive in a single payload transfer.

Table 4.2: Relevant VS Probe and Commit controls fields for frame-based formats

The endpoint direction determines which parameters are negotiable and which are fixed by either the host or the device. For a IN endpoint, the device proposes values for all but 3 parameters: **bmHint**, **bFormatIndex** and **bFrameIndex**, which must be proposed by the host. Some of the parameters are not modifiable by the host.

Figure 4.2 represents the whole UVC video stream initialization. Arrows represent the direction of the data transaction [1]. The host starts by setting parameter values in the Probe control, then it checks if the device accepted them using a GET_CUR. The host can set a parameter to zero to indicate that the device must propose a value. In this example, the device accept the parameters at the first try so no negotiation takes place. If the device had returned different parameter values from those the host had set, the host would have had 2 choices: accept the values proposed by the device and continue with the procedure, or send another SET_CUR to the Probe control with different values, followed by another GET_CUR. In order to avoid negotiation loops, the device has to return parameters with with a lower data rate.

Once the host knows that the device has accepted the parameters, it has to set them in the Commit control with a SET_CUR request. Finally, the host selects an interface alternate with an endpoint corresponding to the stream, e.g. a high bandwidth endpoint for a high bit-rate stream. This endpoint choice is chosen by the host, without negotiation.

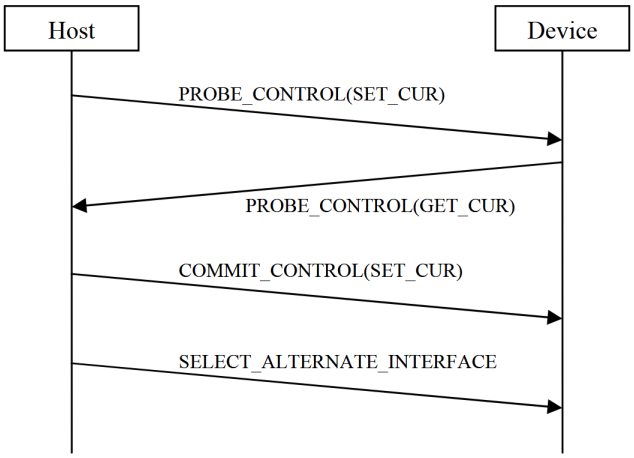

Figure 4.2: UVC Parameter Negotiation
Source: [57]

[1] All USB 2.0 transfers are started by the host, regardless of the data transaction direction.

4.5 UVC Video Stream

Video data is either streamed on an isochronous or bulk endpoint. It is embedded in UVC packets which contain two parts: a payload header and the payload data. Each USB (micro) frame transfer consists of one such UVC packet. In cases where multiple transactions per microframe are possible, the UVC header is only present in the first transaction, and the others only contain data.

In the UVC context, a video sample is an encoded block of video data that can be decoded into a video frame. For formats which use temporal redundancy to compress frames, a video sample is not enough information to form a video frame. For formats which do not, like MJPEG or uncompressed video, a sample stores en entire frame. A video sample is made up of multiple data payloads transfers.

Figure 4.3 depicts a high-speed UVC stream, with 3 transactions per microframe.

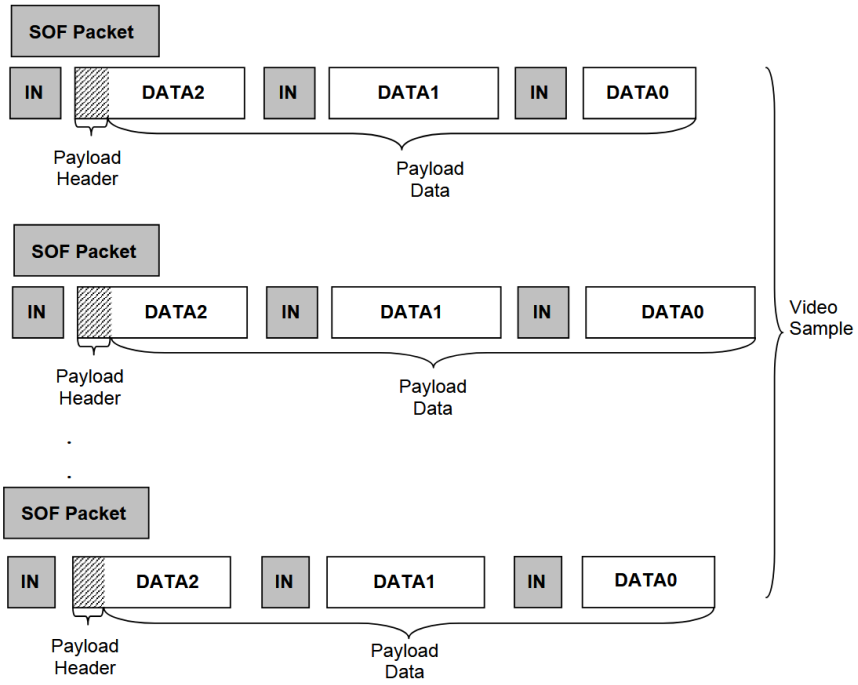

Figure 4.3: UVC Streaming
Source: [57]

The different video formats that can be transmitted using UVC all have format-specific descriptors and different payload header supported fields. These are described in documents separated from the main specification, like the MJPEG format for instance [58].

Table 4.3 details the header fields that are relevant to manage a basic UVC video stream.

Offset	Field	Size (B)	Value	Description
0	**bHeaderLength**	1	Number	Payload header length in bytes, including this field.
1	**bmHeaderInfo**	1	Bitmap	D0: Frame ID, toggles every time a new video sample is sent D1: End of Frame, set if this packet is the last of a video sample D6: Error, set if there was an error during the transmission. The host can learn the cause of the error by accessing the Stream Error Code Control.

Table 4.3: Relevant UVC payload header fields for basic video streaming
Source: [57]

There are 2 types of UVC data payloads defined in the standard:

- Frame-based formats, which do not embed frame boundary information in their stream. This information has thus to be conveyed using UVC packet headers. These formats notably include uncompressed video and MJPEG.
- Stream-based formats, which do embed frame boundary information in their stream. This information can thus be omitted and is optional. These formats notably include MPEG-based formats.

Chapter 5

EHCI Standard

The Enhanced Host Controller Interface (EHCI) is a standardized [59] register-level interface for USB 2.0 hosts. It is only used in conjunction with a USB 1.0 or 1.1 peripheral in the case where a USB 2.0 hub is located in between them, thanks to the use of split-transactions. This standard has to be detailed in order to understand the USB host driver implementation of Renesas, discussed in section 6.4. Considering the UVC application bandwidth needs, only the USB 2.0 details of EHCI are relevant and will be detailed in this chapter, with a particular focus on isochronous and interrupt transfers. Some relevant details of the Renesas implementation for the RZ/A2M are also mentioned in this chapter.

The EHCI interface defines three memory spaces :

- PCI Configuration Space, not implemented in the USB H/W module as no PCI bus is present in this LSI
- Register space, or I/O space, which consists of memory-mapped I/O registers, located inside the H/W module
- Schedule Interface Space, described by the specification as, "typically memory allocated and managed by the eHC Driver for the periodic and asynchronous schedules." For instance, in the Renesas implementation for the RZ/A2M, it is a memory space allocated in the on-chip RAM, and accessible by the USB H/W module via DMA.

USB transfer requests are managed with different data structures according to their type : an isochronous transfer request use an Isochronous Transaction Descriptor (iTD), while interrupt and bulk requests use a Queue element Transfer Descriptor (qTD) located inside a Queue Head (QH) . Those structures are initialized by the software controller and stored in the Schedule Interface Space.

5.1 iTD

The iTD structure is used to manage isochronous transfers. Figure 5.1 describes every fields of the iTD data structure.

The first 32-bit word is used to reference the periodic list element, e.g. an iTD or QH, associated with the transfer to execute during the next microframe of the same frame. These fields will be detailed in section 5.3.

The 8 next 32-bit words represent 8 micro-frame transaction slot, each to be executed at the

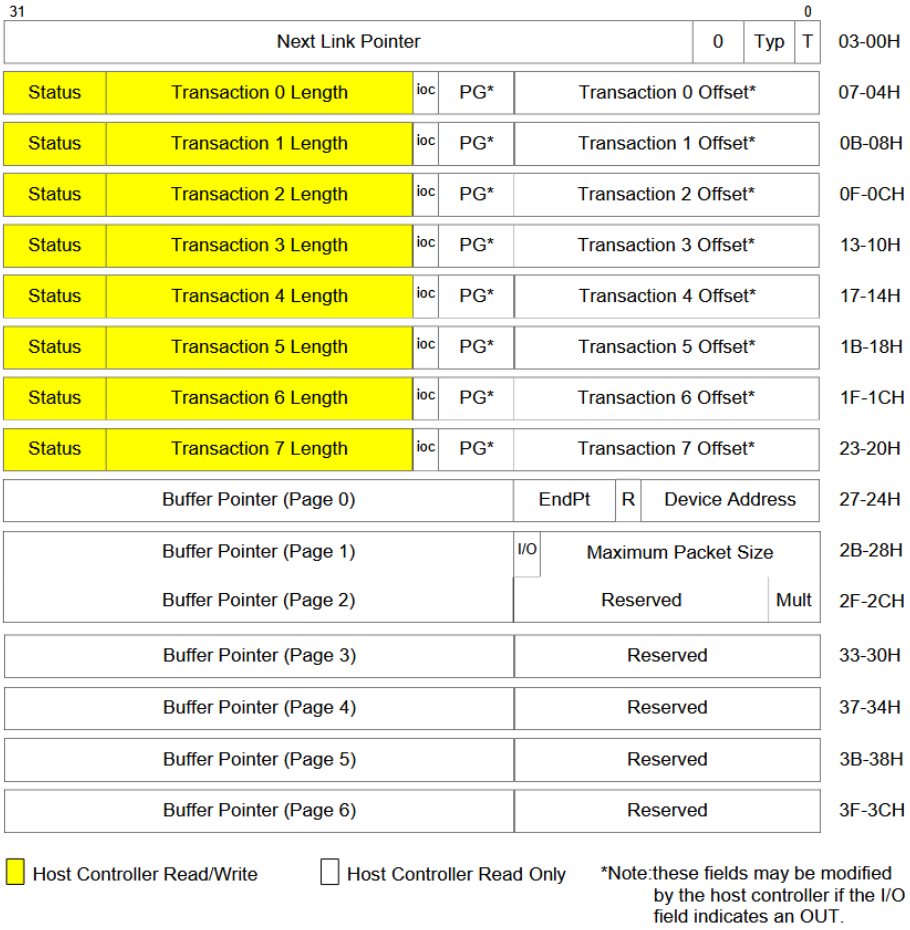

Figure 5.1: iTD organization
Source: [59]

corresponding micro-frame (e.g., transaction 0 with micro-frame 0). Each slot include the following fields :

- Status, used to indicate if the transaction is to be executed or not, as well as the following transaction errors : data buffer error (i.e. transaction buffer underrun or overrun, mostly caused by high memory access latency), babble (i.e., more bytes received than transaction length), and transaction error (i.e. USB protocol errors)
- Transaction length, which either represent the number of data bytes the controller will send (OUT endpoint), or the expected number of received bytes (IN endpoint), to be updated with the number of actually received bytes. This field is a bit misnamed as it stores the length of all transactions executed during a given microframe. As found in equation 3.5, the maximum allowed value is 3072.
- Interrupt On Complete (IOC). The hardware module will generate an interrupt if this flag is set. In the Renesas implementation on the RZ/A2M, it is only set for the last transaction to execute in the iTD.
- Page select (PG), used to indicate on which buffer page the transaction buffer is located.

- Transaction offset, which indicates the offset inside the current page buffer where the transaction buffer begins

The last 7 32-bit words are used to store 7 buffer page pointers, as well as some characteristics of the used endpoint. The buffer page addresses are 4K aligned, which means the 12 least relevant bits are always zero. These zeroes are omitted in the iTD to free up some bits, which are used to store endpoint data instead, as detailed on figure 5.1. The 20-bit page pointers are used in combination with the 12-bit transaction offsets to compute the physical memory address where the transaction data should be written.

Buffer pages must be virtually contiguous, because they will be logically interpreted as a whole by the software. However, they can be physically non contiguous, which is easier to allocate in the RAM. Accordingly, the EHCI software controller has to detect when a transaction overflows a buffer page, and assign the next transaction on the next page at the correct offset. On the other hand, the hardware module has to detect when it has reached the end of one buffer page, and start writing to the physical address corresponding to the next page.

In the worst case scenario, the transaction lengths are all maximum, at 3072 bytes, and the assigned buffer begins at offset 4K - 1. The first transaction thus begins at offset 4K - 1 on page 0, essentially wasting the first page. However, the 6 remaining pages are large enough to store the 8 full transactions. Indeed, 6 page buffers can store $6 * 4 * 1024 = 24576$ bytes, and 8 transactions with maximum MPS amount to $8 * 3072 = 24576$ bytes.

This 7th page thus allows to ease the requirements on the transaction buffer, which can be non aligned thanks to it. This does not mean that some memory is wasted however: naming X the first transaction offset, it implies that every bytes of the first page located before X and every bytes of the last page located after X do not belong to the allocated transaction buffer. This 7th page is not a costly addition to the standard as transactions spread across two pages management must be supported anyway, and the iTD structure has more than enough unused bits to store the additional pointer.

The endpoint-related fields stored in the buffer pages lower bits are : the endpoint number, the device address, the direction (IN or OUT), the MPS, and the multi field. The MPS is limited to 1024 bytes instead of 3072 because it is used in conjunction with the Multi field, i.e. the number of transactions per microframe, which is limited to 3. The rest of the unused bits are reserved for future use, to be specified in a new version of the standard if needed. In the case where the used endpoint is of IN type, i.e. in the case where the host receives data through the endpoint, only the status and transaction length fields are modified by the hardware module, while the others are just read.

As detailed previously, an iTD can store up to 8 micro-frame transfers, which is equivalent to one frame's worth of transactions. The maximum amount of data that can be transferred during one frame was obtained in equation 3.7: $24576B/\text{frame}$. However, the EHCI implementation provided by Renesas only handle 1 transaction per micro-frame instead of the maximum 3. Reusing equation 3.5 with $max(N_{t/\mu f}) = 1$ as only change, and equation 3.7, we obtain as maximum isochronous bandwidth supported by this EHCI implementation:

$$max(D_{\mu f}) = max(N_{t/\mu f}) * max(\text{MPS}) = 1 * 1024 = 1024B/\mu frame \tag{5.1}$$

$$max(D_f) = N_{\mu f/f} * max(D_{\mu f}) = 8 * 1024 = 8192B/\text{frame} \tag{5.2}$$

$$max(B) = 8.192MB/s \tag{5.3}$$

5.2 qTD

The qTD structure is used to manage control, bulk and interrupt transfers. It is stored inside a QH structure, described in figure 5.2.

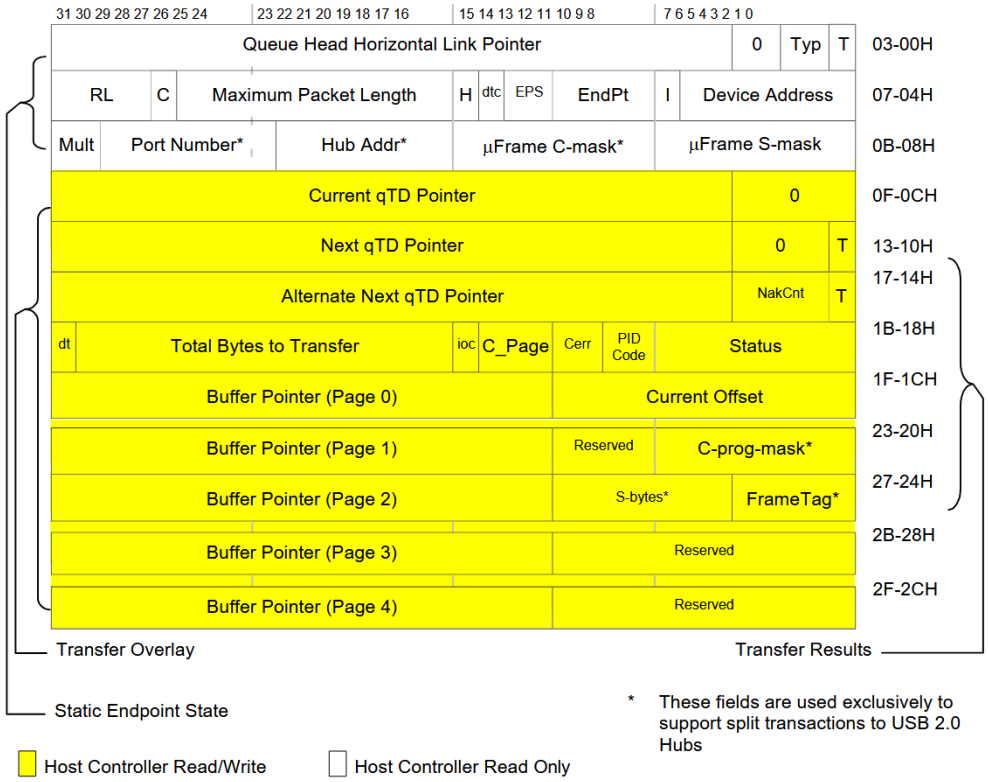

Figure 5.2: QH organization
Source: [59]

The QH is split into two sections. The first is the "Static Endpoint State", which stores information about the next link as well as the used endpoint using the first 3 32-bit words. The second section, the "Transfer Overlay", stores transfer-related data in 2 fields : a pointer to the current qTD (0x0C - 0x0F), and copy of said qtD (0x10 - 0x2F). This copy serves as execution cache, and results must thus be written back into the original qTD instance once the transfers have been completed.

QH fields will not be discussed in detail as the focus of this section is on isochronous transfers. However, it is important to note that the first 32-bit word, which reference the next linked element, allows the QH to be placed either in the periodic or the asynchronous list. These fields will be detailed in section 5.3 and section 5.4.

5.3 Periodic Schedule

The periodic schedule manages time sensitive transfers, and is thus always executed before the asynchronous one. It handles isochronous and interrupt transfers via the periodic frame list: an array of elements containing physical memory pointers to transfer descriptors. The physical memory address of the array root is stored in the PERIODICLISTBASE register, and the current frame

index is stored in FRINDEX. FRINDEX is implemented as a simple N-bit counter, incremented every micro-frame (125 μs) and with N being 11, 12 or 13 bits. The 3 least relevant bits represent the current micro-frame index, and the remaining bits represent the current frame index. A periodic list can thus store 2^{N-3} elements, which gives: 256, 512 or 1024 elements. The address of the current list element can be derived as follows:

$$periodic\_element\_addr = PERIODICLISTBASE[31:12] << 12 + FRINDEX[12:3] \quad (5.4)$$

Figure 5.3 is a diagram which represents the organization of the schedule. On the left, the FRINDEX and PERIODICLISTBASE registers are combined to compute the current list element address. In the center, it can noticed that every list element point to an iTD. On the right, multiple interrupts QH are chained, with different polling rates.

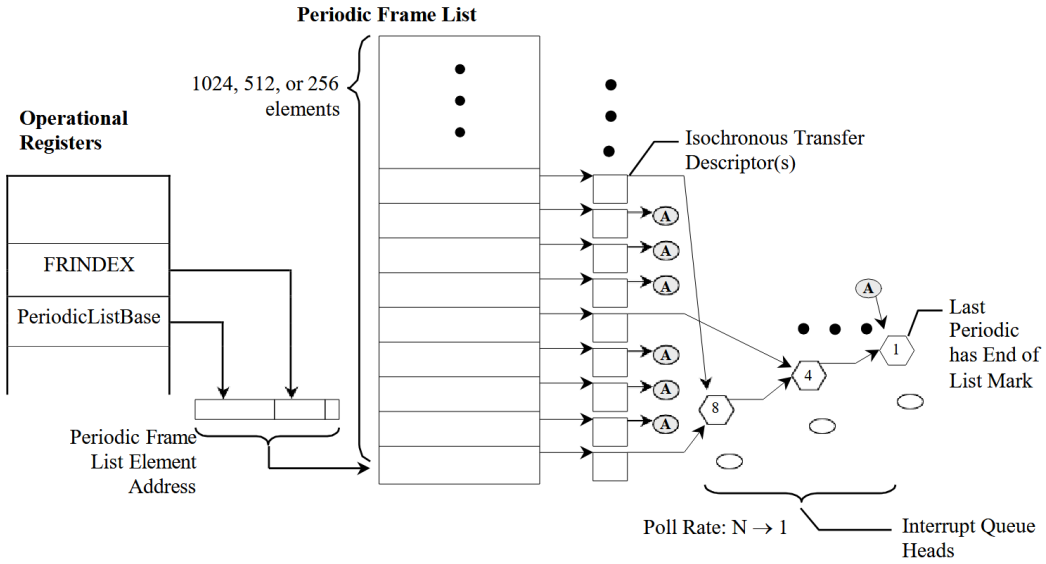

Figure 5.3: EHCI periodic schedule organization
Source: [59]

Periodic list elements possess 3 fields : a 27-bit pointer to a transfer descriptor, a descriptor type (e.g., iTD or QH), and a termination flag to indicate if it is the last element of the current frame. Each time FRINDEX is incremented, the element corresponding to the current frame is checked by the hardware module. If it is an iTD, the transfer corresponding to the current micro-frame is executed if the active flag is set. If non set, the next element is checked thanks to the next link pointer.

If the considered list element is a QH, the transfer is executed if the current FRINDEX correspond to the micro-frame number stored in the S-mask field of the QH. If the termination flag of the list element is set, it implies that no periodic transfer is pending in the current micro-frame, and that bandwidth is available for asynchronous transfers during this micro-frame. This allows the controller to use every micro-frame, which maximizes bandwidth, while still respecting the time sensitive transfer deadlines.

5.4 Asynchronous Schedule

The asynchronous schedule manages transfers without any time constraints. It manages control and bulk transfers via the asynchronous frame list : a simple circular list of QHs, which are linked together using the link pointer in their first 32-bit word. Each QH may point to a qTD to handle, qTD which could also point to other qTDs if multiple transfers are pending. The schedule uses a round-robin architecture to split the bandwidth across multiple endpoints : the controller first executes the transfer detailed in one qTD of the current QH before switching to the next QH, even if more qTDs were linked. However, the park mode allows the controller to execute multiple transfers from one QH before swapping, in order to maximize bus usage. Furthermore, to ensure timely execution of synchronous transfers, the controller must stop executing the asynchronous schedule upon the reception of an end of micro-frame packet, and execute the periodic list instead. The ASYNCLISTADDR register stores the address of the next QH to handle, and is thus a kind of list index. Figure 5.4 is a diagram which represent the ASYNCLISTADDR register and the circular list.

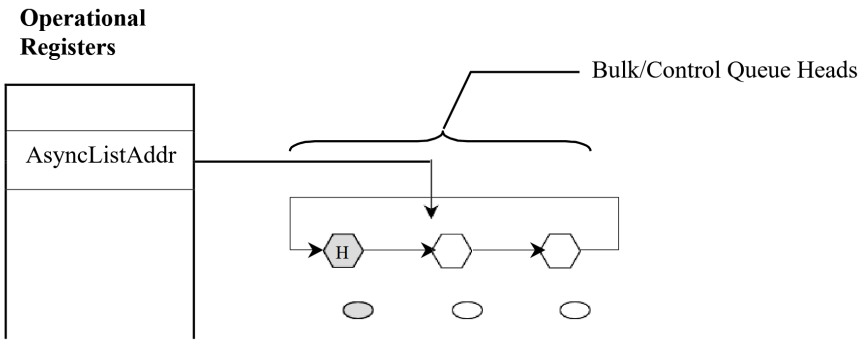

Figure 5.4: EHCI asynchronous schedule organization
Source: [59]

Chapter 6

Implementation on Development Board

This chapter describes the implementation of video streaming on the development board. The UVC application was first implemented on this board instead of the arDent board for more convenient development. Indeed, using the devolvement board allows convenient testing and integration of sample codes, since they were designed for it. For instance, the microcontroller pin assignments already correspond to the circuit board, so no need to change them.

Furthermore, debugging is easier as the code is small. Indeed the project was started from a code sample, with the unnecessary parts removed. In contrast, no sample codes exist for the arDent board: the actual code used for the Head-Mounted Display application would have had to be used. Moreover, removing necessary parts to ease debugging is not trivial, as many software modules depend on each other.

First the board itself is described, then the UVC application.

6.1 Board Characteristics

This section describes the relevant characteristics of the development board.

- 1 RZ/A2M microcontroller, described in section 6.1.1
- 1 USB-C receptacle
- 1 USB mini-B receptacle, used as power supply
- 1 USB micro-B receptacle, used as serial port to allow debug printing on PC terminal
- Multiple RAM modules: not used since none is present on the arDent board [1]
- 1 display output board: it transfers the output image of the Video Display Controller (VDC) present on the RZ/A2M, to a display connected through HDMI.
- 1 CoreSight 20 connector: used to program the board using the J-Link Segger debug probe

The development board is actually composed of two boards, SUB [60] and CPU [61], but it is not relevant to mention the location of components. Furthermore the display output board provided with the development board kit is separate. It is connected using a 50-Pin MIL connector.

[1] Apart from the RAM module inside the microcontroller naturally.

6.1.1 RZ/A2M Architecture

The RZ/A2M microcontroller is a Large Scale Integration (LSI) chip from the company Renesas, i.e. an integrated circuit which consists of many transistors (up to billions) and is capable of implementing complex features on a single chip. Multiple peripheral modules are present on the RZ/A2M LSI [62]. Here is a list of those that are relevant for this application:

- 1 Cortex A-9 processor as Central Processing Unit (CPU)
- 2 USB 2.0 modules, which can each act as either host or peripheral
- 1 JPEG Codec Unit (JCU), which can encode and decode JPEG images
- 1 Video Display Controller (VDC)
- 1 Direct Memory Access (DMA) controller. It implements the DMA feature described in section 6.2.2.
- 4 MB of RAM

The on-chip peripherals are directly connected to the Cortex A-9 processor thanks to a bus interface, as shown on figure 6.1. Section 6.2.1 describes how the processor can interact with those peripherals, and section 6.2.2 details how the peripherals can directly access the memory without using the processor.

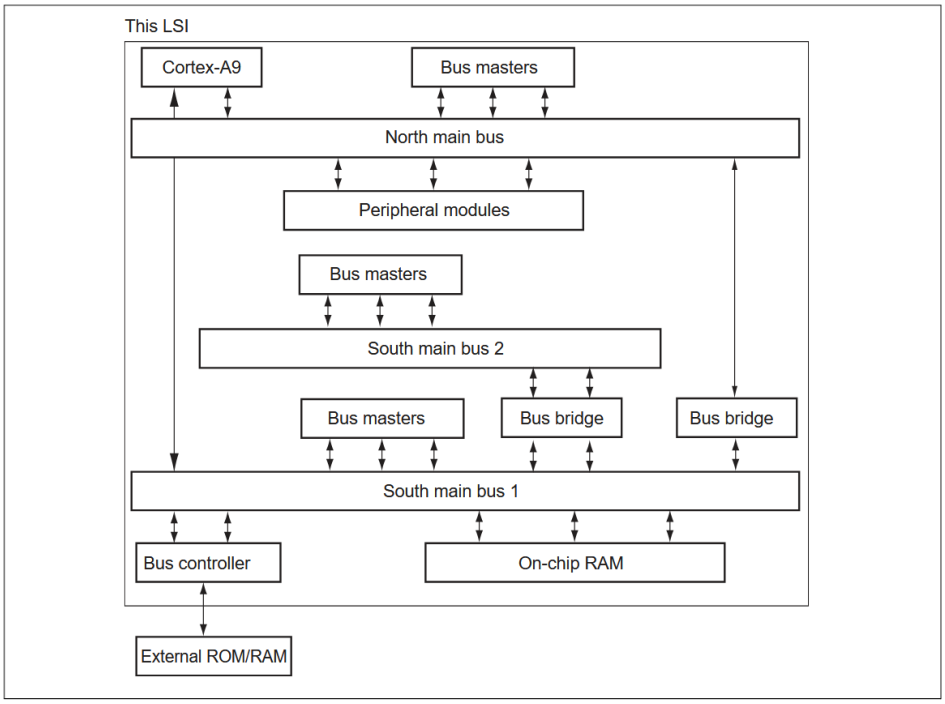

Figure 6.1: LSI bus diagram
Source: [62]

6.2 On-chip Peripheral Connectivity

This section details how the CPU and the on-chip peripherals can interact with each-other.

6.2.1 On-chip Peripheral Register Access

Some of the peripheral registers must be accessible by the CPU for read and write operations, and are thus mapped to physical addresses in the peripherals I/O page, and accessible via bus interfaces. The accessible registers of a given module are organized in a continuous address block, located at an address fixed when the LSI was designed. These registers can be be organized with the C language using a volatile structure. The volatile keyword instructs the compiler to not skip any memory read when optimizing the code, because the register values might have been modified asynchronously by the hardware module. The use of a structure allows to easily organize the registers, as shown in listing 6.1, which uses the registers of the USB Host module of the USB port 0 as an example. The use of union structures and bit-fields allows to choose between accessing the whole register or just a specific field. Registers fields are described in the RZ/A2M specification [62].

```
struct st_usb00
{
    union
    {
        unsigned long LONG;
        struct
        {
            unsigned long CBSR:2;
            unsigned long PLE:1;
            unsigned long IE:1;
            unsigned long CLE:1;
            unsigned long BLE:1;
            unsigned long HCFS:2;
            unsigned long :1;
            unsigned long RWC:1;
            unsigned long :1;
            unsigned long :21;
        } BIT;
    } HCCONTROL;
    ... // other union structures
}
#define USB00     (*(volatile struct st_usb00     *)0xE8218000)

USB00.HCCONTROL.BIT.PLE = 1; // Set PLE to 1
USB00.HCCONTROL.LONG = 3; // set PLE to 1 and all the other fields to 0
```

Listing 6.1: USB H/W register access

6.2.2 Direct Memory Access

DMA is a feature that allows on-chip peripherals to access memory independently from the CPU. DMA memory accesses are handled by the DMA controller, also located in the microcontroller. The peripherals must use physical addresses because virtual addresses are handled by the Memory Management Unit (MMU) located inside the CPU [63]. To avoid having to manage the cache state, it is preferable to use uncached memory for DMA.

DMA is highly beneficial for this UVC application because large amounts of data have to be accessed by the JCU and the VDC in particular. Thanks to DMA, the CPU can issue a request to a peripheral, and execute other non-related operation while waiting for this request to finish. A hardware interrupt can then be used to notify the CPU that the work has been performed.

6.3 UVC Application Architecture

The code of this UVC project is divide into multiple parts, located in different folders. Only the folders and files relevant to understand the project are shown on figure 6.2.

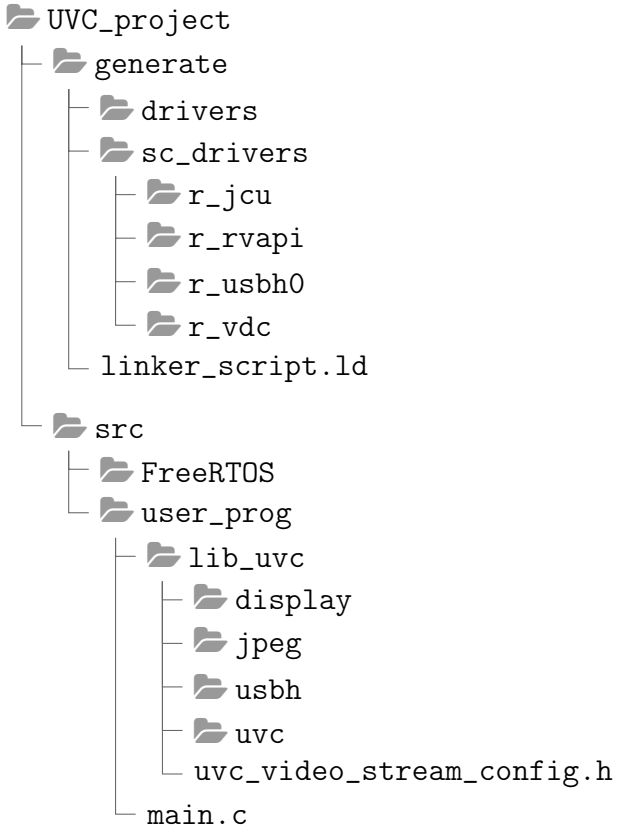

Figure 6.2: UVC Project Organization on Development Board

Here follows a short description of each file and folder:

- generate: regroups the files generated and modifiable by the Smart Configurator [2]. Not all

[2]A graphical interface that is used to modify projects on Renesas microcontrollers. It handles pins, clocks, drivers...

sub folders are shown here.

- generate/drivers: regroups the core drivers of the microcontroller, which handle caches, interrupts, GPIO ports,...
- generate/sc_drivers: regroups on-chip peripheral drivers
- generate/sc_drivers/r_jcu: the JCU driver, used by the *lib_uvc* JPEG package discussed in section 6.6
- generate/sc_drivers/r_rvapi: it is a video-related API, used by the *lib_uvc* display package described in section 6.7
- generate/sc_drivers/r_usbh0: the USB host driver, described in section 6.4. It is used used by the UVC and USBH packages discussed respectively in sections 6.8 and 6.5.
- generate/sc_drivers/r_vdc: the VDC driver, used by the *lib_uvc* package described in section 6.7.
- generate/linker_script.ld: defines the memory location of specific code sections or data structures
- src/FreeRTOS: the FreeRTOS OS. The Smart Configurator does not modifies it.
- src/user_prog: regroups user-made code. In this case, there is only *lib_uvc* and a main.c: they form the UVC application.
- src/user_prog/lib_uvc: regroups the different packages needed for the UVC application
- src/user_prog/lib_uvc/display: integrated sample code from Renesas. Described in section 6.7.
- src/user_prog/lib_uvc/jcu: integrated sample code from Renesas, which implements JPEG decoding. Described in section 6.6.
- src/user_prog/lib_uvc/usbh: integrated sample code from Renesas, which initializes the USB ports. Described in section 6.5.
- src/user_prog/lib_uvc/uvc: the implemented UVC driver, described in section 6.8
- src/user_prog/lib_uvc/uvc_video_stream_config.h: this header file defines parameters used in multiple *lib_uvc* packages, namely video frame dimensions and number of bytes per pixels
- src/user_prog/main.c: contains a main function that notably initializes FreeRTOS, then calls the initialization function from usbh that initialize the *lib_uvc* packages. This file is not part of *lib_uvc*.

6.4 USB Host Driver

This section describes the USB host driver *r_usbh0* provided by Renesas which can be added using the Smart Configurator, and the modifications that had to be performed to suit the UVC application. This driver is not complete: a few initialization functions had to be taken from a sample code. This is described in section 6.5 about the USB Host module.

6.4.1 USB Host Driver Architecture

Renesas has developed and provides a USB host driver that serves as interface between the USB 2.0 Hardware (H/W) module present on the microcontroller chip, and USB class drivers. An application note [64] describes the driver features, high-level architecture, Application Programming Interface (API) and application-specific configuration options. Figure 6.3 is a block diagram that details the driver architecture, with the host driver colored in yellow.

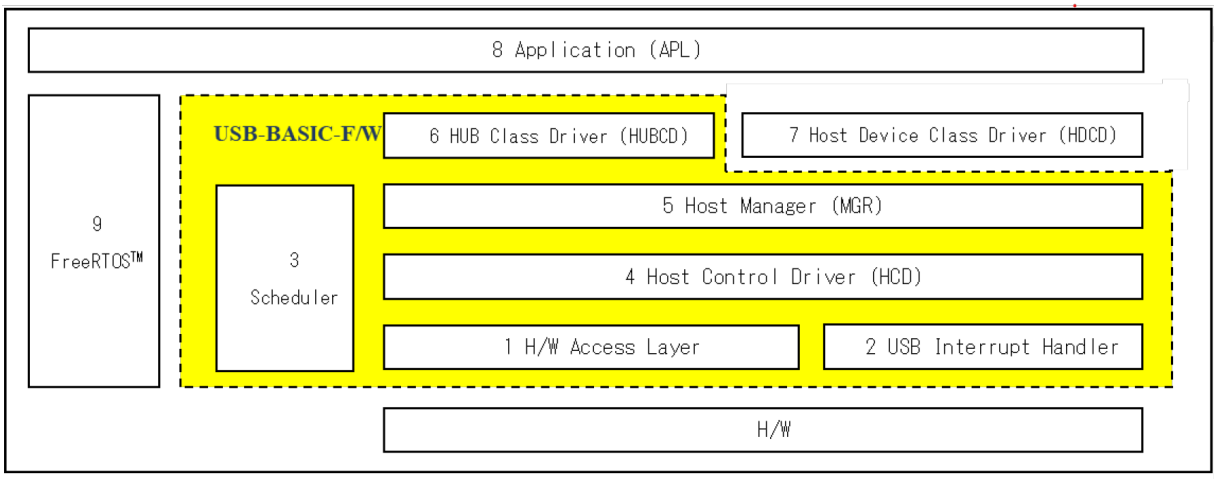

Figure 6.3: USB host driver architecture
Source: [64]

Here follows a description of each layer, from high to low level.

6.4.1.1 Application

The main application software. For this driver to work, it must contain:

- The driver initialization functions, in order to initialize the USB port, the USB hardware module, the host driver tasks, and the needed class-specific drivers.
- A RTOS task which must loop over different functions in a round-robin fashion, including the scheduler described in section 6.4.1.6. In the case where no RTOS is present, it must also run the tasks which have received mail using the mailbox system (again, see section 6.4.1.6). It does so by calling the task main functions directly. This loop will be denoted as the main round-robin loop in the following sections.

In the USB-related sample codes provided by Renesas, the main round-robin loop also includes a serial task which uses the USB driver for demonstration purposes.

6.4.1.2 HDCD

The Host Device Class Driver is responsible for handling class-specific requests. It consists of a set of callbacks and a task, whether implemented in the main round-robin loop or as a RTOS task. It is not part of the host driver, but they interact closely with each other. Indeed, during enumeration, the host manager task has to check whether the connected device is supported or not by the installed class drivers. It first simply checks the class ID, then calls a callback function , provided by the class driver, to perform a more complex check. Other callbacks can also be provided by the class driver, to be called when the following events occur : device configuration terminates, device gets detached, device enters low power mode and device leaves low power mode. Thanks to the use of callbacks, the host driver does not need to be modified in order to be compatible with user-made class drivers.
The UVC driver described in section 6.8 is an example of a class driver.

6.4.1.3 HUBCD

The Hub Class Driver is a type of class driver which implements the following features :

- Device state transition callbacks, as described in section 6.4.1.2
- State management of every downstream USB port of the hub, as well as every devices attached to the hub.
- The start of the device enumeration sequence, i.e. : upon device connection, the HUBCD requests the hub to reset it, and retries until it succeeds. Finally, the HUBCD asks for the device descriptor, and the host manager takes over the enumeration from that point on.
  Before being able to make USB transactions with the device, such as asking for the descriptor, the HUBCD has to register the device address, port and speed to the HCD.

The HUBCD is included within this host driver because unlike the other class drivers, it cannot be implemented with just a few callbacks. Indeed, it directly interferes with the enumeration process. Furthermore, a dedicated RTOS task exists to handle hub-specific requests.

6.4.1.4 MGR

The Host Manager (MGR) is an intermediate layer between the HCD and the class drivers. It implements the following features :

- Class driver registration, as described in section 6.4.1.2. It is necessary for enumeration.
- State management of the USB port and every connected devices
- Device enumeration and configuration. Those two processes are mandatory before the device accepts any class-specific requests, and relatively complex since multiple USB transactions and a state machine are needed. It is thus sensible to implement them inside this generic host driver instead of requiring each class driver to implement them.

The MGR is implemented using a dedicated RTOS task divided in two serial tasks : the HCD task runs first to handle USB request receptions, then the MGR serial task is called to act accordingly.

6.4.1.5 HCD

The Host Control Driver (HCD) is the main layer of the host driver, responsible for managing the USB transfer requests. It is implemented through an API and a serial task located inside the MGR RTOS task. The HCD implements the following features :

- Control transfers, with result notification
- Data transfers (bulk, isochronous and interrupt), with result notification
- Forced termination of data transfers
- Error detection in all USB transfers
- Transfer retry on error for control and bulk transfers, as per the USB 2.0 specification [56].
- State management of the USB port and USB port reset handling
- Low power mode commands (suspend and resume)
- Handling of USB hardware interrupts

The HCD does not directly interact with the hardware module. Instead, it serves as an abstraction layer over the Host Controller Interface (HCI), which is the software component responsible for directly controlling the hardware module. To do so, the HCI accesses USB H/W module registers and also manage structures stored in the RAM that are accessed by the H/W module using DMA. Multiple standard HCI implementations exist, each supporting different USB speeds. Two are present in this driver : the Open Host Controller Interface (OHCI), which handles USB 1.0 and USB 1.1, and the EHCI, which handles USB 2.0. Given the bandwidth requirement of this project, USB 1.0 and 1.1 are irrelevant, so only EHCI is of interest here. It is discussed in detail in chapter 5.

6.4.1.6 Scheduler

This USB driver uses mailboxes to transmit requests between different processes. Mail data is first written into a global array before being handled by the scheduler, in descending order of priority of the destination mailbox. If FreeRTOS is present, the scheduler will unblock the task using the FreeRTOS queue associated with the receiving task. If no RTOS is present, the scheduler will just raise a flag so that the receiving task will be called in the main round-robin loop. In both cases, the receiving task will read the mail located in the mailbox data structure. As mentioned earlier, the scheduler task is located inside the main round-robin loop.

6.4.1.7 USB Interrupt Handler

Every hardware interrupts generated by the USB 2.0 hardware module are routed to the same interrupt handler function. This handler then passes the event code to the HCI task, using the mailbox system. This thus creates a delay in the event handling as the scheduler managing the mailboxes is executed in the main round-robin loop.

6.4.1.8 FreeRtos

The FreeRTOS Real-Time Operating System (RTOS). It can be noted that this driver is also runnable bare-metal, i.e. without any RTOS, thanks to the use of conditional compilation directives.

6.4.1.9 H/W Access Layer

The Hardware Access Layer of the USB 2.0 hardware module is the interface used by the HCD uses to interact with the hardware module. It consists of all the memory-mapped I/O registers of the H/W module that are accessible by the CPU, as described in section 6.2.1. This layer is well described in the hardware manual of the microcontroller [62].

6.4.1.10 H/W

The USB 2.0 hardware module, an on-chip peripheral located inside the microcontroller package. It implements the OHCI and EHCI standards.

6.4.2 Modifications to the USB Host Driver

6.4.2.1 Configuration Parameters

The UVC driver was designed starting from the sample code example implementing the USB Mass Storage Class (MSC). Some configuration parameters from the generic host driver had to be modified to suit the needs to the UVC driver, and are listed here :

Table 6.1: Modified USB driver parameters

USBH0 parameter	Initial value	New value	Comment
CFG_CLASS	0	2	Used to initialize MSC (0) or HID drivers (1) tasks. Both must be disabled here.
CONFIGSIZE	256	1028	Size of the device USB descriptor. UVC devices have large descriptors.
EHCI_NUM_ITD	4	1	Number of iTDs, statically allocated. Only one is used by the UVC driver.
EHCI_NUM_-SITD	4	0	Number of siTDs, statically allocated. None is used by the UVC driver.
MAXPIPE	32	16	Max number of active pipes, and pending transfers. The UVC driver uses two while running (isochronous + interrupt). The lower the less memory allocated.
MAXHUB	5	1	Max numbers of connected hubs. Assuming no external hub is used, 1 internal hub inside peripheral is allowed. Using 1 allows to reduce MAXDEVADDR.
MAXDEVADDR	12	2	Max number of drivers and connected devices. Only 2 needed : hub + UVC. The lower the less memory allocated.

6.4.2.2 Scheduler

As mentioned in section 6.4.1.6, the scheduler provided by Renesas is implemented as a serial task inside the main round-robin loop, and is checked at a high frequency. This is a wasteful usage of processor time since the scheduler might be checked even if no mail is pending, in which case it will not do anything. Furthermore, this round-robin architecture mixed with RTOS tasks offer very poor time guarantees. Indeed, the main task is used as idle task, which is never blocked. It must thus have a lower priority than every other tasks because the RTOS would never yield the processor to a task if a higher priority task is unblocked. A serial task located inside the round-robin, such as the scheduler, has as maximal latency bounded by a sum of 2 terms : the sum of latencies of all other tasks in the round-robin, as well as the longest time interval in which others RTOS tasks are active. This time interval might be longer than the sum of all RTOS task latencies because some tasks may be executed multiple times.

This waste of processor time and poor time guarantee are not problematic in the Mass Storage Class sample code for instance, as it only contains a few USB-related tasks and most importantly no time sensitive processes. Indeed, the only time constraints are the USB transfers handling, located

on the hardware level. This architecture allowed Renesas to make the driver both runnable with or without FreeRTOS, using a few conditional compilation directives. For instance, the OSless scheduler does not send messages to tasks using the FreeRTOS queue system to unblock them, it simply calls the main functions of said tasks directly.

However, this scheduler implementation can not be used as is in a more complex project which includes time consuming tasks and periodic tasks, for the reason mentioned previously. The CPU processing power wastage is especially undesirable on an embedded device, because of the finite energy supply.

The scheduler had thus to be modified to fix those issues. Ditching the round-robin architecture removes the OSless capabilities of the driver, but they are not needed anyway since the application uses FreeRTOS. The only scheduler feature is mailbox handling, and two architectures were evaluated to implement it in a more efficient way :

- No scheduler architecture : tasks directly use the FreeRTOS queue system to exchange data and notify each other. It is more efficient than using an intermediate scheduler, but the driver has to be extensively modified. Indeed, each RTOS task must be modified to read "mail" data from the FreeRTOS queue directly, instead of the mailbox data structure. Furthermore, the MGR and HCD share the same RTOS task, and have to be separated into two separate tasks so that they can each wait for messages without blocking the other. Furthermore, the HCD task has to run before the MGR task.

- Scheduler in RTOS task architecture : the scheduler is located inside its own RTOS task, and is only executed when mail has to be handled. The macros that send mail thus have to be modified to notify the scheduler task. A lightweight counting semaphore is used to keep track of the needed number of scheduler task execution. The FreeRTOS direct to task notification mechanism [65] is used to implement this semaphore, in order to reduce the latency between a mail request and its treatment. Indeed, a direct notification sets the state of the targeted task to pending directly, without the need for the OS to check any intermediate object.

The second architecture was chosen, although less efficient, because it involves less modifications in the driver code. Keeping modifications to a minimum allows easy integration of updates to the original driver, reduces development time and the odds of introducing bugs.

In both case, the interrupt handler has to be modified, because the high priority of the USB hardware interrupt prevents the use of RTOS functions, such as sending a message on the FreeRTOS queue. Two fixes are possible : either the interrupt priority is decreased, or a second lower priority interrupt needs to be triggered by the first one. The downside of low priority interrupts is that they are disable in critical sections, which raises their maximum latency. However in the case of USB hardware interrupts, extremely low latency is not needed as it is the hardware module that handles the tight time constraints of the USB 2.0 standard. The first fix was thus chosen as it is simpler, without compromising the correct functioning of the USB driver.

6.4.2.3 EHCI

The EHCI implementation from Renesas had to be modified, whether to add some functionalities or fix a bug. 3 functions were modified, located inside r_usbh0_hehci_transfer.c:

- *usbh0_hstd_ehci_make_isochronous_request()*: this function from the EHCI layer is responsible for filling an iTD with the request-related data and linking it to the periodic list. During the iTD allocation, the driver searchs for an already existing iTD with corresponding endpoint, address and direction. If no fit iTD exists, a new one is created.
  The modification consist in bypassing the search, so that a clean iTD is always used . This ensures that iTDs only contain information about the current transfer, not the previous ones.

- *usbh0_hstd_ehci_init_itd()*: this function is used by *usbh0_hstd_ehci_make_isochronous_request()* to set some of the iTD fields, such as the transaction buffer-related fields. Renesas left a bug in this function, more specifically in the part that initializes the transaction offset fields. It will be detailed in the following paragraphs.

  In their implementation of the iTD structure, Renesas decided to include HCD-related fields, not described by the EHCI standard. These fields are only used by the EHCI software controller, not the hardware module. One of these fields is the transaction buffer, implemented as an array of 8 elements, storing data for each micro-frame transaction. This transaction buffer is used to compute the buffer pages and transaction offsets.

  The faulty line is written in listing 6.2. It is located inside a loop that writes the offset for each of the 8 iTD transaction.

```
1 R_MMU_VAtoPA((uint32_t)(tmp_bufferadrs & 0x00000FFF)
2               ,p_itd->transaction[n].bit.offset);
```

Listing 6.2: Faulty iTD transaction offset setting

  tmp_bufferadrs is the virtual address of the transaction buffer stored inside the iTD structure. It is incremented by the max transaction size (1024) each loop. *R_MMU_VAtoPA()* is a function that takes as input a virtual address, and a physical address pointer. Upon execution, the physical address corresponding to the input virtual address is written to address pointed by the input pointer, and an error code is returned. The function declaration is written in listing 6.3.

```
1 e_mmu_err_t R_MMU_VAtoPA(uint32_t vaddress, uint32_t * paddress)
```

Listing 6.3: *R_MMU_VAtoPA()* function declaration

  The intended effect of the faulty line (listing 6.2) is that the 3 least relevant bytes of the physical address corresponding to the current tmp_bufferadrs are written in the offset field. However, one can notice two issues with this line. First, the logical AND operation should not be applied to the virtual address since this limits the input virtual address to the 12 least relevant bits. This is nonsensical because virtual addresses between 0x0 and 0x03FFFFFFF belong to the "Chip select 0" memory space on the MSC example MMU, not the RAM. Second, *R_MMU_VAtoPA()* takes a pointer as second input, but in the faulty line the transaction offset bit-field is directly used. Once again, this does not make any sense : the address of the offset field should be given as argument. However, it is not possible to use addresses of bit-fields

using the C language, since the unit of addresses is the byte, not the bit. An intermediate variable thus has to be used.

Here is the solution used to correctly set the transaction offset, written in listing 6.4.

```
1 R_MMU_VAtoPA(tmp_bufferadrs,&tmp_addr);
2 p_itd->transaction[n].bit.offset     = tmp_addr & 0x00000FFF;
```

Listing 6.4: Correct iTD transaction offset setting

This error was spotted because it directly impacted the transfer buffers: data would be found at strange offsets and some packets were cut in half. This seemed to indicate timing issues at first, as if the host was missing the first transmitted bytes or was not issuing enough request and was being overwhelmed by data. The Renesas support was contacted but could not identify the issue either [66]. Eventually, the error cause was found and solved as explained earlier.

- usbh0_hstd_ehci_transfer_end_itd(): it is called after a USB hardware interrupt caused by an iTD transfer completion or failure. This function copies data from the transaction buffer to the buffer used to make the HCD transfer request, parses the status field, and calls the callback attached to this HCD request.

  Two additions were made to this function. First, the iTD is cleared entirely since they are not reused anymore in *usbh0_hstd_ehci_make_isochronous_request()*. Second, transaction lengths are added as small headers before the transactions content. Indeed, little information is returned to the driver which has made a HCD transfer request. The callback attached to the HCD request only contains 3 inputs: the total transfer length, the transfer status and the initial request handler, which contains a pointer to the transfer buffer. This buffer contains the data of each transactions, without the low-level USB headers, in one raw stream of bytes.

  Distinguishing the transactions is not trivial as transaction lengths can be lower than the MPS and the transaction content differ from one USB class to another. For instance, a UVC video stream contains one header per micro-frame transaction. Unfortunately, this header does not contain the transaction length, in order to limit the protocol overhead. Trying to locate the header of each transaction by looking for specific recurring values is too hazardous and could lead to false header identifications.

  The solution proposed by the Renesas support [66] is to use only one transaction per iTD, but this limits bandwidth and is therefore not suitable. Indeed, the HCD transfer initialization and termination takes non negligible time. In the end, the solution used is to append the length of each transaction just before the data of said transaction, as a kind of header. It is stored as a little endian 2-byte number, since transactions can reach up to 1024 bytes with this EHCI implementation. Accordingly, HCD transfer buffers must be larger to accommodate this added header.

6.5 USB Host Package

This section describes the USB Host package of the *lib_uvc*.

6.5.1 Sample Application

Renesas provides 3 sample applications related to USB and using FreeRTOS: a Human Interface Device (HID) class host, a Communications Device Class (CDC) function, and a Mass Storage Class (MSC) host. The MSC sample application [67] was chosen as a base to implement the whole UVC application because the two classes handles significant amounts of data, and it uses the USB host role.

6.5.2 Sample Code Integration

Little was kept from the MSC sample code in the end. The noteworthy code integrations include a utility function that translates a physical memory address to virtual address, used by the EHCI implementation described in chapter 5, and some initialization functions: they initialize some pins used by the USB host driver as well as some register fields of the USB hardware module.

This package also contains a function that regroups every needed initialization functions from the *lib_uvc* packages, *usbh_uvc_init()*. This function is the only one that must be executed in order to initialize the whole UVC library.

6.6 JPEG Decoder Package

This section describes the JPEG codec application provided by Renesas, and how it was integrated into a package to suit the UVC application.

6.6.1 Sample Application

Renesas provides a library [68] with an API to control the JCU hardware module present on the microcontroller, *r_jcu*. They also provide a sample application [69] which uses this API to implement two use-cases: encoding and decoding. In the first case, an example JPEG stored in memory is decoded then encoded back to JPEG. In the other case, more relevant for the UVC application, the sample JPEG is decoded then displayed on a screen connected through HDMI.

R_JCU_SampleDecode() is the function from the sample code which implements JPEG decoding and image display. It has no arguments. It initializes the JCU, decodes a JPEG frame and stores the uncompressed frame into another buffer. Finally, it initializes the VDC driver, and uses it to display the frame on the connected screen.

It can be noted that since the JCU uses DMA to access the JPEG and decoded frame buffers, the CPU is not needed for most of the decoding process. The sample code frees up the CPU during those time periods by using semaphores to block the task handling the decoding. The task is unblocked once the semaphore is set, which is done by the interrupt handler which manages JCU hardware interrupts.

6.6.2 Sample Code Integration

The only code snippet taken from the sample code is the function that implements decoding taken: *R_JCU_SampleDecode()*. A few modifications had to be made however, detailed in this section. The JCU driver library *r_jcu* is imported and left unmodified.

- In the sample code, the JCU hardware module is initialized in the decoding function. Since this initialization is only needed once, the whole initialization section is removed and placed in its own function, *R_JCU_INIT()*, to be called at device initialization or upon JCU error.
- *R_JCU_SampleDecode()* now takes as arguments the address of the buffer storing the JPEG, the JPEG size, and the address of the buffer to use to store the decoded image.
- This sample code displays the decoded picture on a screen. Only one display buffer is used however, which can sometimes cause graphical glitches if this buffer gets displayed by the VDC while its content is being written. This is not an issue for this sample code as the sample picture is only displayed once, so glitch probability is low and their effects are negligible.

  However, this does be an issue if this code is used to display video. This display-related part was thus removed entirely, and another sample code was used to display images (see section 6.7).

The modified *R_JCU_SampleDecode()* function can be concisely summarized as follows: it decodes a JPEG frame buffer passed as argument and stores the uncompressed frame in another buffer passed as argument.

6.7 Display Package

This section describes the VDC application provided by Renesas, and how it was integrated into a package to suit the UVC application.

6.7.1 Sample Application

Renesas provides a library with an API to control the VDC present on the microcontroller, *r_vdc*. They also provide a sample application [70] which uses this API to display the video stream of a MIPI CSI-2 camera. Figure 6.4 depicts the sample code architecture. The main library used it *r_rvapi*: it contains an API which can be used to control the VDC (using *r_vdc*), a MIPI camera and the Capture Engine Unit (CEU). The CEU is responsible for writing the video data acquired in memory via DMA, which allows to free up the CPU during this operation. The VDC management is the only relevant part of this library as the physical interface used for the UVC application is USB, not MIPI CSI. Indeed, the UVC driver is the one managing the camera, and the EHCI hardware module handles data acquisition and writing into the memory, also via DMA.

Only the VDC handling part of the sample code is of interest: it is entirely located within one file, *r_bdc_lcd.c*. This file implements display of video frames, as well as overlay of text on top of displayed images. Unlike the display-related code in the JPEG sample application, two display buffers are used in this one. When one buffer is being displayed, the other can be modified without the risk of causing any graphical glitch. Once the idle buffer has been written into, an API function is called inform the VDC of the address of the buffer which is now to be displayed.

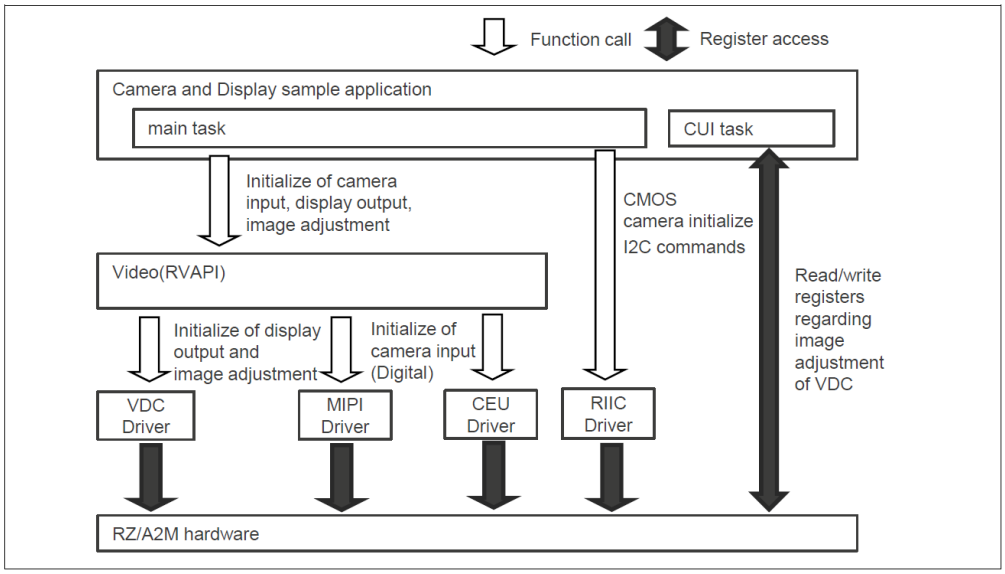

Figure 6.4: Camera and Display sample application architecture
Source: [70]

6.7.2 Sample Code Integration

The *r_vdc* library is simply imported and left unmodified. The *r_rvapi* library is also imported, but only the VDC-related files are kept. The files of *r_rvapi* related to the MIPI, CEU and SPEA drivers are discarded since they are not relevant for the UVC application.

The *r_bdc_lcd.c* source file is the only one needed for the UVC application. As overlay-related code is not needed for the UVC application, it was commented out using conditional compilation directives. This saves memory as the text overlay is done using an additional pair of display buffers. This code was not deleted in case this functionality is needed in the future. A few overlay-related files are thus also present in the modified sample code.

Here follows a description of the functions used in the UVC application:

- *R_BCD_LcdInit()*: initializes the VDC and hardware interrupt. It can be noted that the frame dimensions and uncompressed format are needed for this initialization.
- *R_BCD_LcdGetVramAddress()*: returns the address of the display buffer currently not being displayed, and which is thus modifiable.
- *R_BCD_LcdSwapVideoBuffer()*: sets a global swap flag informing that the currently idle display buffer is ready to be displayed.
- *IntCallbackFunc()*: this interrupt handler is called when a VDC hardware interrupt is triggered. It handles the VSYNC: if the display buffer swap flag is set, the new frame contained in the idle buffer is displayed. Otherwise, no action is required.

Some modifications had to be performed on *r_bdc_lcd.c*:

- *R_BCD_LcdInit()*: some initialization parameters are changed to match the values used in the JPEG sample application described in section 6.7. For instance, the uncompressed format of the image to display is now RGB656.

- The two buffers used to store the uncompressed images to display are now bigger by $0x10000$ (65536) bytes. This addition comes from the JPEG sample code, which requires it.
- The image stride macro definition had to be modified. The image stride is the space taken in memory by a pixel line, as described on figure 6.5. It depends on the number of bytes used to encode a pixel, and on the end-of-line padding. This padding is used to accelerate memory access to the image buffer, by aligning each each line in memory. In the UVC application, the image buffer is provided by the JCU, and no padding is used. This fits the UVC application well since memory is more important than processing speed. The stride is thus simply computed as the number of bytes per pixels, multiplied by the the image width.

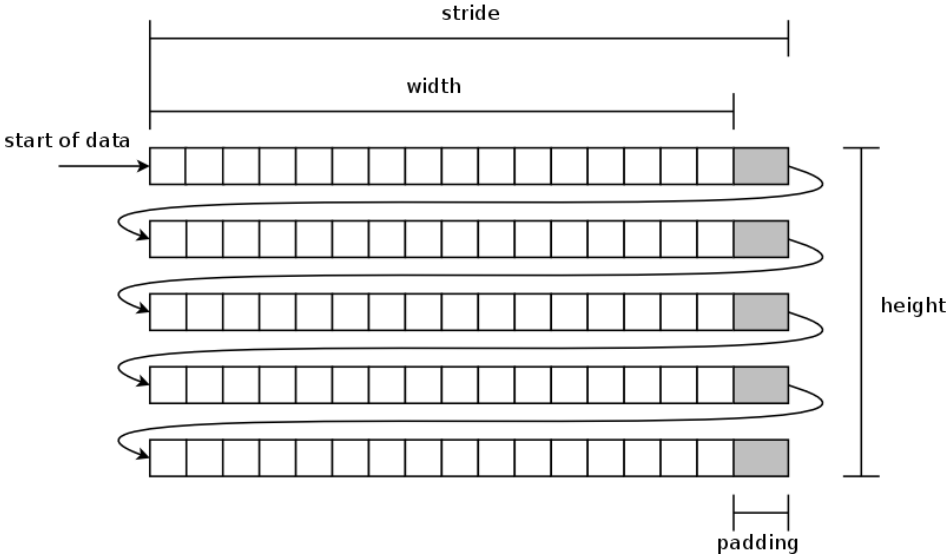

Figure 6.5: Image Stride
Source: [71]

6.8 UVC Driver Package

This section describes the UVC package, which implements a USB Video Class (UVC) host driver. It implements basic video streaming and display using the other packages of *lib_uvc*. The Mass Storage Class sample application [67] was used as initial base, but little was kept.

6.8.1 File Organization

The UVC driver is organized into multiple .c and .h files, which are detailed in this section.

uvc_driver.c This source file contains the driver main task function, as well as the driver registration function to allow the interfacing with the USB host driver, as described in section section 6.4.1.2.

uvc_api.c This source file contains public functions that are used in uvc_driver.c, as well as private variable and utility functions.

uvc_rtos.c This source files contains the function which initializes the UVC driver task and mailbox. It also contains task-related parameter definitions, i.e. the stack size and task priority.

uvc_if.h This header is the only one that has to be imported in order to run the driver. It contains the definition of the only initialization function needed to run the UVC driver.

uvc_local.h This header contains every function, macro and parameter definition needed by the other files of the UVC driver. It is not meant to be included by files non part of this driver.

uvc_video_stream_config.h This header contains parameter definitions of frame characteristics that are needed for both the JPEG package and the UVC driver for buffer initialization purposes, i.e. the frame dimensions and the number of bytes per pixels of the uncompressed frame format. These parameters are set at compilation and not dynamically acquired by parsing the camera UVC descriptors because every data structure of the *lib_uvc* packages are statically allocated at compilation. This allows easier debugging, since they are thus no runtime issues because of a full memory heap.

This header is not part of the UVC driver itself, but of the *lib_uvc* library.

6.8.2 Finite State Machine

This driver mainly executes and reacts to USB requests. The same USB request response can trigger a different reaction from the driver depending the past requests executed, i.e. the state of the driver. The Finite State Machine (FSM) architecture is thus well-suited for this application. Figure 6.6 describes the FSM used, and this section details the events and actions related to each state. For clarity, each state is named after the main action it contains. Some actions are to be executed whatever the state entry or exit, and are noted inside the states. It is important to note that self-transitions triggers both state exit and entry actions.

Here follows a simplified overview of the states, before a more in-depth description in the following sections. Upon initialization, the driver waits for a UVC device to connect in **WAIT_FOR_DEVICE**. Once a camera is attached, the driver accesses sets the VS interface in **SET_CONTROL_IF**, and negotiate streaming parameters during the **GET_PROBE** and **SET_PROBE** states. The negotiated parameters are then set in **SET_COMMIT**, and the right VS alternate setting interface is set during **SET_STREAMING_IF**. Then, the video stream is started in **START_STREAMING**. The host waits to have received enough data for a frame during the **STREAMING** state. Finally, the received frame is displayed in **DISPLAY_FRAME**, and the driver restarts streaming by transitioning to **START_STREAMING** again.

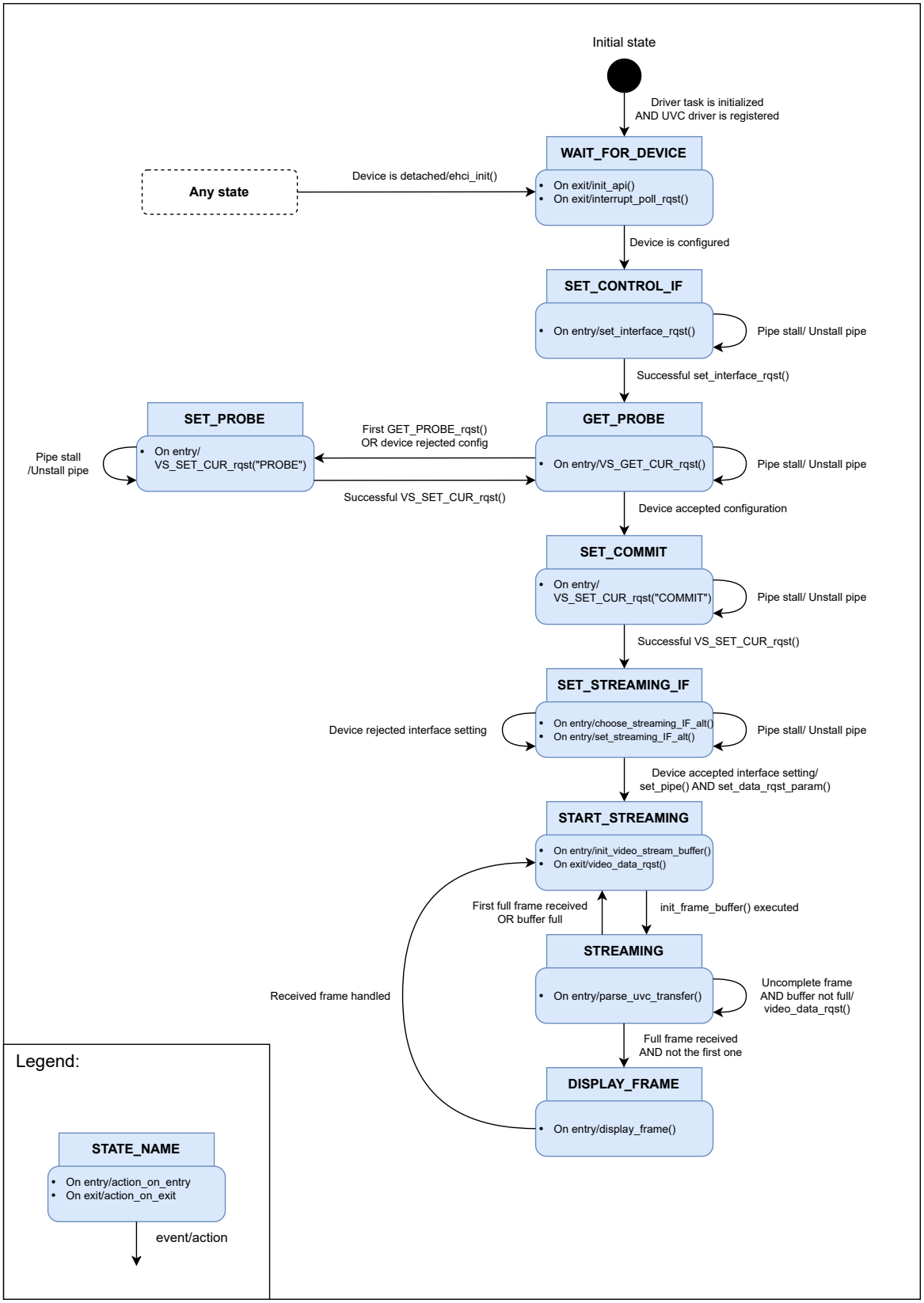

Figure 6.6: UVC driver state diagram

6.8.2.1 WAIT_FOR_DEVICE

This is the initial state upon driver initialization : the driver task is initialized and the UVC driver is registered to the USB host manager as described in section 6.4.1.2, but no device is yet configured and linked to the driver.

Upon device attachment, the host manager calls 2 UVC-related callbacks:

- *uvc_class_check()*, is called during the configuration stage described in section 3.4.3. Indeed, once the host driver manager (see section 6.4.1.4) has received the full device descriptor, it has to choose a driver to manage the device. In order to do so, the host driver first parses the device descriptor to search for an interface with the same class than the currently examined driver. If the class matches, it then calls the driver class check callback for further analysis.

  In this driver, the class check function *uvc_class_check()* parses the descriptor to check if at least one video control and video streaming interface are present, otherwise the driver does not accept to manage the device. Furthermore, it sets up the control interrupt pipe that is used for error interrupts. It also stores data for future use, such as: the descriptor, device address, ID number of the control and streaming interfaces and supported UVC version. Finally, it also chooses the frame format and dimensions according to the parameters defined by the user: the maximum dimensions below or equal to what the user defined are chosen. This choice can not yet be communicated to the device however: it will be during the VS negotiation.

- *uvc_driver_configured()*, called by the host driver once the device is configured. It initializes the UVC API global variables using *uvc_init_api()*, polls the control interrupt endpoint with *uvc_interrupt_poll_rqst()* and changes the state to **SET_CONTROL_IF**.

The UVC driver task does not do anything in this state : it is blocked until a mail arrives, i.e. a USB request related to the UVC driver has to be treated. Indeed, the only USB request that could terminate in this state is the data transfer request on the control interrupt pipe, in case the device sends error data immediately, but this is not expected under normal operating conditions however. If the interrupt transfer ever terminates because the device sent data error, the request will be restarted regardless of the current state.

It must be noted that in case a device linked with this driver is detached, a class-specific callback is called: *uvc_driver_detach()*. This function sets the driver state to **WAIT_FOR_DEVICE**, whatever the previous state, and resets the EHCI driver using *usbh0_hstd_ehci_init()*. Upon new device attachment, the *uvc_init_api()* that will be called at the exit of **WAIT_FOR_DEVICE** will clean the driver state, which is necessary in order to handle the newly connected device. The EHCI reset cleans the EHCI state, especially the periodic schedule, as well as any iTD or QH in use. As only the camera ought to be connected during video streaming, this EHCI reset does not perturb any other device.

6.8.2.2 SET_CONTROL_IF

Once configuration is done, the host has to set an interface in order to access the needed functions. Since the VC controls do not have to be modified in order to have a basic video stream (see section 4.1), there is no need to set the VC interface. The VS interface thus has to be set.

This state is triggered by the *uvc_driver_configured()* function, which is called upon the end of device configuration. Apart from the actions related to the previous state, it also contains one function related to this state: *uvc_set_interface_rqst()*. This function send a request to set the device interface to the video streaming one, with 0 as alternate setting. Once this request is accepted by the device, VideoStreaming controls can be now accessed by the host, and the state machine transitions to **GET_PROBE**.

6.8.2.3 GET_PROBE

This state handles a part of the VS parameter negotiation described in section 4.4. The entry action is executing the function *VS_GET_CUR_rqst()*, which sends a GET_CUR request to the VS Probe control. Once the request succeeds, the driver parses the received content of the VS Probe control using *uvc_parse_VS_config()*. Relevant parameters for basic video streaming are described in table 4.2. Multiple state transitions are possible depending on the received parameters:

- If this is the first GET_CUR, the host has to set the parameters it would like in the Probe commit, taking the values it has just received into account. In this driver, an IN endpoint is used, so there are only 3 fields that must be set by the host. To simplify the driver, only these required parameters are set by the host, and the others use the values proposed by the device. These 3 parameters are: the hint bitmap (set to 0 as no additional parameters to those 3 is fixed by the host), the format ID and frame ID [3]. The driver state transitions to **SET_PROBE**.
- If the parameter values received differ from those the host had previously set, the negotiation continues: the host has to set other values using the device feedback. The driver state transitions to **SET_PROBE**.
- If the parameter values received are identical to those of the host, the negotiation is over and the the host has to set those values in the Commit control. The driver state transitions to **SET_COMMIT**.

6.8.2.4 SET_PROBE

This state continues the VS parameter negotiation. The entry action is executing the *VS_SET_CUR_rqst()* function with the VS Probe control as argument, to be used as control selector value. This function sends a SET_CUR request this control, using the parameters values defined in the previous state. Once the request has succeeded, the state is set back to **GET_PROBE** in order to check if the device accepted the parameters.

6.8.2.5 SET_COMMIT

This state concludes the VS parameter negotiation. The associated entry action is the *VS_SET_CUR_rqst()* function with the VS Commit control as argument. Once this request has succeeded, the driver transitions to **SET_STREAMING_IF**.

[3] The format and frame IDs were chosen with *uvc_class_check()* during state WAIT_FOR_DEVICE

6.8.2.6 SET_STREAMING_IF

This state handles the last stage of the VS interface configuration: the endpoint choice, which is communicated to the device by setting one of the VS interface alternate settings. The first entry action is the **uvc_choose_streaming_IF_alt()** function: it continues to parse the configuration descriptor tree starting from the VS interface with alternate setting set to 0. This function checks every VS endpoints and chooses the one with the endpoint which has the highest bandwidth. However, as summarized by equation 5.1, the maximum bandwidth allowed by the EHCI implementation used is limited by a max MPS of 1024 and a max numbers of transactions per microframe of 1. These limitations must be respected when choosing an endpoint.

Once an interface alternate setting has been chosen, it is set by using *uvc_set_interface_rqst()* with the alternate setting ID as argument. The driver then checks if the device is indeed using the correct alternate setting with the *uvc_get_IF_alt_rqst()* function: if it is not using the right one, the driver re-enters this state in order to retry setting the interface.

If the device is using the right alternate setting however, the state switches to **START_STREAMING**, after executing two functions. The first is *uvc_set_pipe()*, which sets up a pipe to the chosen streaming endpoint. The second is *uvc_set_data_rqst_param()*, which sets the data request parameters which do not vary from a request to another. Indeed, setting those parameters in the request function itself would slow it slightly, which is not desirable for real time streaming.

6.8.2.7 START_STREAMING

This state consists in initialization of UVC API global variables related to parsing, and executing the initial data transfer request. The functions used to achieve these tasks are respectively *uvc_init_video_stream_buffer()* and *uvc_video_data_rqst()*. This state was added for clarity purposes: since it only contains 2 actions to execute successively upon entry, it could easily be merged into others.

6.8.2.8 STREAMING

This state implements the process of executing data transfer requests, parsing them, and stopping once a full video frame has been received or the video stream buffer is full. The first action is parsing the received video data using *uvc_parse_uvc_transfer()*. There are three possible outcomes for that parsing:

- The frame has not been fully received yet and there is space left in the video stream buffer. Another data transfer request is queued via *uvc_video_data_rqst()*, and a self-transition to the same state occurs.
- The frame has not been fully received yet but the video stream buffer is full. The buffering has to restart from the start, and the state thus transitions back to **START_STREAMING**.
- The entire frame has been received. If it is the first full frame ever received since the device got attached, it is discarded by setting the state to **START_STREAMING**. Indeed, the first frame ever sent is often corrupted. If it is not the first one however, the driver state transitions to **DISPLAY_FRAME**.

Transfer parsing is not trivial because UVC packets might contain errors: packet are not resent upon error with isochronous pipes. Furthermore the camera might send multiple packets without payloads in burst when no video data is ready to be sent. The parsing function *uvc_parse_uvc_transfer()* was thus designed to be robust. For instance, both the Frame ID (FID) and the End-of-Frame (EoF) fields are checked. If the FiD toggles but no EoF flags was set in the previous packets, it implies that the last frame was not received entirely and that it must thus be discarded.

6.8.2.9 DISPLAY_FRAME

This state only contains one action, upon entry: *uvc_display_frame()*. This functions starts by extracting the video frame from the UVC packet stream stored in the video stream buffer, i.e. copying only the payloads and discarding the headers. The buffer to use to store this decoded frame is determined using the *R_BCD_LcdGetVramAddress()* function from the display package (see section 6.7.2). It then decodes the frame using the *R_JCU_SampleDecode()* function from the JPEG package (see section 6.6.2), and displays it if there are not JCU error using the *R_BCD_LcdSwapVideoBuffer()* function from the display package (see section 6.7.2). The state then transitions back to **START_STREAMING**.

The JCU is configured to output RGB565 frames, and the VDC is configured to accept them. It can be noted that thanks to conditional compilation directives, uncompressed video is also supported by *uvc_display_frame()*. The video frame is simply directly extracted into the display buffer, without any decoding stage. However, the VDC parameters are not modified: if the received uncompressed frame uses a different format with the same numbers of bytes per pixels, it will still work but the wrong colors will be displayed.

6.8.2.10 Pipe STALL handling

As mentioned in section 3.3.2, STALL conditions can be returned during a control request. The standard advises that control pipes only return protocol STALLs and no functional STALLs. However, a badly designed device could potentially return functional STALLs anyway.

In this UVC driver, when the EHCI driver returns a STALL for a control transfer, the STALL is cleared as if it were functional by using the API of the USB host driver. The current state is then entered again, which means the entry action is executed again. As control transfer request are only used as entry actions, the driver will loop in a state and retry the problematic control request until it succeeds.

6.8.3 Streaming Architecture

As described in section 6.8.2.8, video steaming was implemented as a serial process. First, the host requests a data transfer using one iTD with 8 microframe transfers. It then waits for the device response, and start parsing the video stream buffer when the whole buffer has been received. Depending on the UVC packet headers, the host either requests another data transfer or display a full frame. The video frame has first to be extracted from the UVC packet stream received before being displayed. It is only after the frame has been displayed that the host requests the next data transfer.

Multiple reasons justified the use of a serial architecture. First, the HCD does not support queuing requests on the same pipe natively. Furthermore the EHCI implementation of Renesas was not designed to allow precise placements on the asynchronous schedule: the first free slot is just used. As a request function only handles one iTD, there is no guarantee that calling it multiple successive times would result in adjacent iTDs in the schedule, since the request takes a non negligible time to be registered.

Another reason is memory usage optimization: there is only one request structure and one iTD needed for instance. Continuing to stream during JPEG decoding would require another video stream buffer. Restarting the stream after the frame is extracted from the UVC packet stream would be feasible however, but the display frame rate is already satisfying and it is not a trivial thing to do. Indeed, it would require the addition of another FreeRTOS task since the UVC driver task is blocked while waiting for the JCU to finish decoding.

Finally, the bandwidth is already sufficient with this serial architecture anyway. Indeed, the transmitted video frames use JPEG compression, so the compression ratio can be expected to be around 10 [72]. The display on the *arDent* board has 854x640p as dimensions and pixels are usually encoded on 2 bytes in uncompressed formats. Streaming at 30 frames per second, the resulting data rate is thus: $\frac{854*640*2*30}{10} = 3.279MB/s$. This is the maximum expected bandwidth need, and it is lower that the maximum bandwidth of this EHCI implementation, which was found to be $8.192MB/s$ in equation 5.3. During testing, the compression ration varied between 11 and 25, which is even better.

Chapter 7

Project Porting to the arDent Board

This chapter details how to port the UVC library to the *arDent* board, and the hardware limitation that prevents it.

7.1 arDent Display Hardware

As was mentioned in section 1.1.2, the *arDent Display* device is separated into 2 interlocked components. One contains the main circuit board, the*arDent* board, which contains the display and the microcontroller. The other one contains the battery board and a USB-C receptacle. These two components are located into 2 different casings, and fit together thanks to a 5-pin magnetic connector.

7.1.1 arDent Board

Here is a list of the relevant components of the *arDent* board for the UVC application:

- 1 RZ/A2M microcontroller
- 1 5-pin custom magnetic connector which can transmit USB 2.0 data signals
- 1 Bluetooth module
- 1 854x640p display

Contrary to the developed board, there is no serial port, which means debug prints have to be handled in another way. The J-Link Segger debug probe can be used for thar purpose.

7.1.2 Current Battery Board

Here follows a list of the battery board main components:

- 1 battery
- 1 USB-C port
- 1 5-pin custom magnetic connector which can transmit USB 2.0 data signals

- 1 fuel gauge

Figure 7.1 depicts the part of the battery board schematic which contains the USB-C connector and the magnetic 5-pin connector. The 3 components in yellow, from left to right, are: the USB-C connector (124019362112A), a protection diode (TPD4E02B04DQAR), and the custom 5-pin connector.

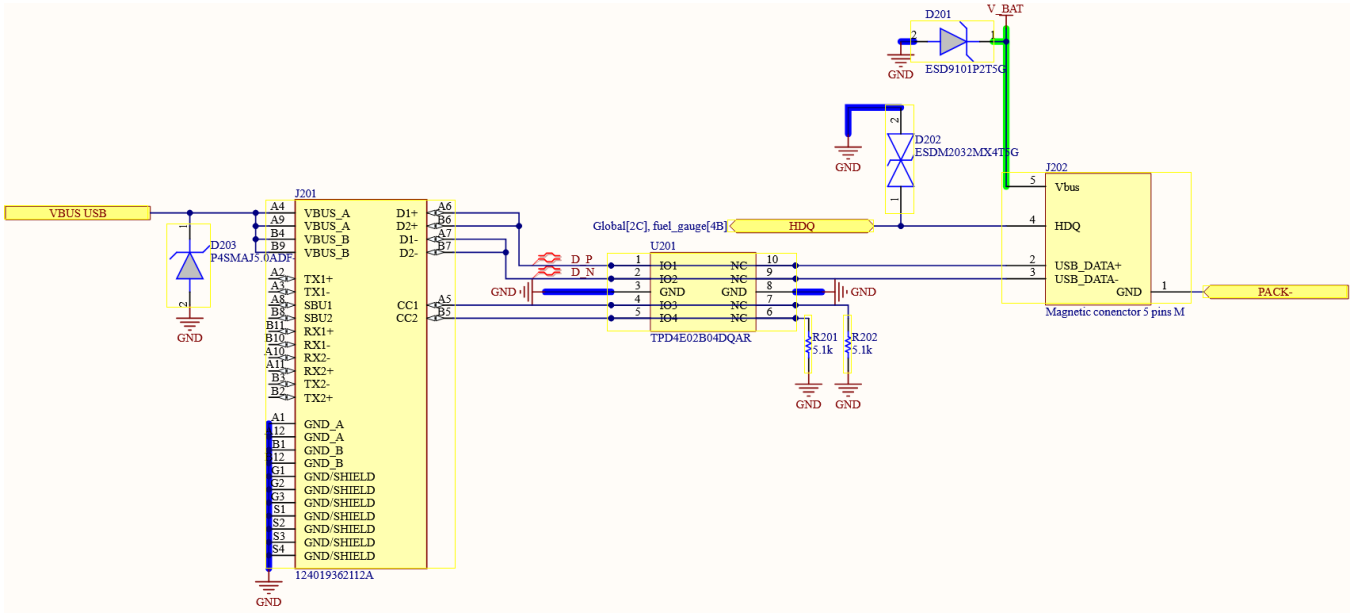

Figure 7.1: Current *arDent* Battery Board
Source: [73]

As the RZ/A2M only contains a USB 2.0 controller, the pins of the USB-C port related to USB 3.0 are ignored, except CC1 and CC2. As was mentioned in section 2.1.14.3, those Communication Channels (CC) pins are notably used to determine the direction in which the port is facing: either upstream (equivalent to USB 2.0 device) or downstream (equivalent to USB 2.0 host). It can be noticed on figure 7.1 that both CC pins are connected to a $5.1k\Omega$ resistor, in a pull-down configuration. However, according to the USB-C specification [1], this is the characteristic configuration of a sink (USB equivalent to 2.0 device). Figure 7.2 illustrates a connection between a source (host) and a sink (device). The source has to monitor the voltage of the 2 CC lines: a voltage drop in either of them indicates that a source was connected. Only one CC line is connected at a time, which allows to determine the cable orientation.

This CC configuration implies that with this battery board, the *arDent Display* can only be used as USB device, and never as host. A port of the UVC host driver discussed last chapter is thus not possible with this battery board.

[1] More specifically section 4.5 of the specification [46].

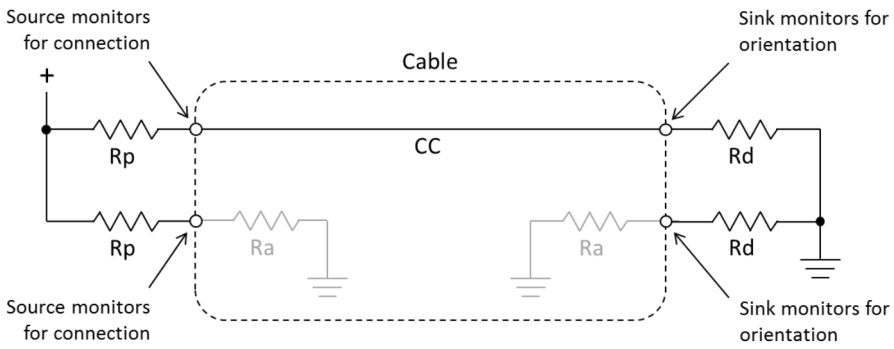

Figure 7.2: USB-C CC connection between a source and a sink
Source: [46]

7.1.3 New Battery Board

In order to allow the *arDent Display* to be used as USB host, a new battery board has to be designed. However, compatibility with the *arDent Board* must be kept, as well as USB device functionality. These are two heavy constraints: the new battery board must implement a Dual-Role-Data (DRD) port i.e, a port which can act as either upstream or downstream, without any contact with the microcontroller, since the 5 lines of the magnetic connector are already used.

Get Your Way hired contractors to design this new battery board, and the first draft schematics were already done at the time of writing this report, although the prototype board was not ready yet. The solution they proposed is to use a dedicated device to handle the CC logic, as well as a power path management device to handle the bidirectional power supply (direction depends on the USB role).

The *TUSB320 USB Type-C Configuration Channel Logic and Port Control* [74] device used in the draft schematic is configured as DRD port acting as a source. This configuration is done via 2 input pins : ID set to high and PORT, which is left unconnected. This static configuration does need any interaction with the microcontroller, which fits the requirement stated earlier. It must be noted that this device only handles the two CC pins of the USB-C port.

7.2 arDent File Organization

This section describes how *Get Your Way* has organized the firmware that can be used on the *arDent Display*.

Get Your Way has developed multiple software applications for the *arDent Display*. The main one is called *app_ardent*: this is the firmware that is used in sold devices. It is quite complex as it contains every functionality developed by *Get Your Way*. Other simpler applications exist, which mostly implement hardware module tests. The entire *arDent*-related code is stored into one repository, *gyw_ardent*: applications are to be compiled through Makefiles that only include the necessary libraries. Applications folders thus only contain application-specific code, and compilation-related files.

Figure 7.3 depicts the simplified organization of the *gyw_ardent* repository.

- generate: regroups the files generated and modifiable by the Smart Configurator, such as peripheral drivers. It is the same as in section 6.3.

- src: contains multiple applications as well as libraries, even though only one of each is shown.
- src/vendor: this folder regroups every library not written by *Get Your Way*. This includes FreeRTOS and USB related libraries for instance.

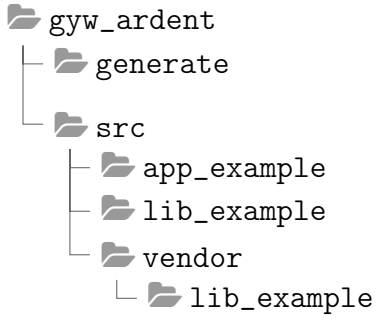

Figure 7.3: *gyw_ardent* repository organization

7.3 UVC Application Integration

This section details the UVC project integration process. Due to time constraints and the issue of the fixed USB role to device (see section 7.1.2), this process could not be performed. This section is thus purely theoretical.

The UVC application can be integrated as library in the src folder. The main initialization function (in usbh package) has to be called at device power-up in order to initialize each package. Some modifications are needed for the packages to work, and are discussed in the following paragraphs.

The first task is to adapt the usbh package. Indeed, the initialization of the USB host driver may need modifications if the USB-related pins on the microcontroller differ from the development board. The USB mass storage device driver already implemented on the *arDent* board can be used as reference. It must be disabled for the UVC application to work however, so that the *arDent Display* is only configured as a USB host.

The following task is to modify the initialization of the VDC done in the display package to suit the screen mounted on the *arDent* board. There already exists a library which implement still image display for the the *arDent* board and that can thus serve as reference.

Concerning the JPEG package, it must be noted that the JCU is already initialized by one of the libraries used in *app_ardent*, so the JPEG package initialization function is not necessary and may be removed.

The UVC package does not need any modifications.

Finally, the linker_script.ld file has to be modified in order to fit the needed data structures into the uncached RAM [2]. However, it must be noted that since most of the data structures needed for *arDent_app* operations are initialized dynamically, and are placed on the memory heap in the cached RAM. This implies that reducing the cached RAM too much will create run-time errors when the heap is full, but none at compilation. A solution could be to also use dynamic memory allocation for

[2]Needed for DMA and on-chip register access

the UVC application, and either using an uncached heap or flushing the cache manually. Another solution could be to disable the graphical interface displayed on screen while no camera is connected, since it is display-related data structures that use most of the memory.

Chapter 8

Conclusion

A state of the art of the physical interfaces used to retrieve video data from a camera has been performed, as well as the determination of the most suited one for a possible new version of the *arDent* board.

In depth summaries about the USB 2.0, UVC 1.0, and EHCI standards have been written, to help *Get Your Way* acquire in-house expertise about them. These summaries focused on the parts relevant to understand USB and UVC host drivers.

Packages implementing video display and JPEG decoding have been implemented starting from sample codes. Furthermore, the USB host driver has been made less computationally intensive by modifying its scheduler. A bug was found and fixed in the register-level EHCI driver, and its maximum data rate was raised by allowing iTDs with 8 microframe transfers to be used.

A firmware implementing the streaming of MJPEG video frames via a USB-C physical interface, and display of said frames on a screen has successfully been implemented on the RZ/A2M microcontroller. The achieved frame rate is satisfying, and any frame resolution is supported, as long as there is enough memory. Furthermore, detachment then re-attachment of the UVC device during operation is supported.

Finally, the porting of the UVC application from the development board to the *arDent* board was analyzed. The hardware constraint of the battery board preventing the porting was identified, and the new board which fixes it was briefly described. Eventually, the steps to perform the porting with the new battery board were detailed.

Although the final porting to *arDent* board could not be performed, the overall project is a still a success: video streaming was implemented on the RZ/A2M, and the porting only consists in changing a few initialization sequences.

8.1 Next Developments

This section lists some improvements and new features to add to this UVC project.

The first obvious next work to perform is to port the UVC application to the *arDent* board, as discussed in section 7.3. As mentioned in the same section, using dynamic memory allocation instead of static would allow to free up some memory used by the UVC application. It is preferable to use a non cached heap to avoid having to flush the cache manually.

Compatibility tests with different cameras could be performed, in order to verify if the UVC driver implemented is universal. It could also be useful to implement some of the camera-related optional features of the UVC 1.0 standard, or the later versions. These optional features include: zoom, focus, exposure time, brightness, contrast,...

Once the project is integrated, a nice feature would be the ability to take photos upon Bluetooth commands. The video stream provided by the camera would allow convenient aiming of the camera. There are 3 ways to implement this feature using the UVC standard [57], the simplest being to extract a frame from the video stream. However the requirement of a double video frame buffer limits the frame resolution. A better approach, optionally supported by the specification, is to stop the video stream and reuse the isochronous endpoint to send a single still frame of higher resolution. In this case, the double video stream buffer can even be reused to store this still frame. The last solution is to keep the video stream going and use an additional bulk endpoint to transmit the still frame. It is also a non mandatory addition to the standard.

Implementing decoding of other video formats would make the UVC driver compatible with more camera models, as well as allowing higher frame resolution. However using the CPU to decode video frames would be inefficient in terms of energy consumption and CPU usage. A better idea would be to use the Dynamically Reconfigurable Processor (DRP) [75] present on the RZ/A2M chip to implement video codecs. As the name indicates, it is a processor that can be reconfigurable during runtime, unlike a FPGA which is configured at compilation. For instance, figure 8.1 shows a performance comparison of a H.264 codec implementation using the RZ/A2M CPU and the DRP, or just the CPU.

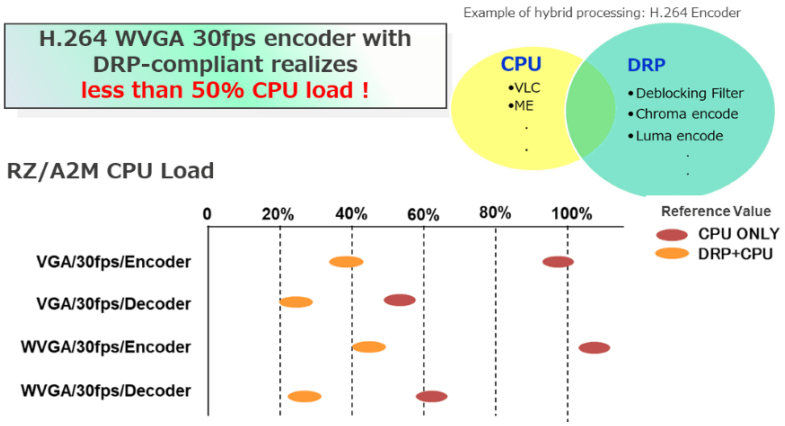

Figure 8.1: Performance of H.264 codec on RZ/A2M DRP
Source: [76]

Chapter 9

Annex

This thesis was realized in collaboration with the company *Get Your Way* [1]. This annex describes the company as well as the internship.

9.1 Get Your Way

Get Your Way is a young company specialized in hands-free picking [1] solutions for the hospital and logistics sector. It is based in Liège, Belgium.

Figure 9.1: *Get Your Way*'s logo
Source: [77]

9.1.1 Products

The company sells multiple products:

- *arDent Display*, a lightweight Heads-Up Display (HMD).
- *ScanPad*, a small keyboard with embedded bar code scanner, attached on the wrist
- *Picking Pilot*, a connector software that links *Get Your Way* devices with Enterprise Resource Planning (ERP) systems. It contains additional features, such as picking list management between multiple users.

The main use of these products is to make picking more convenient and error-free. Indeed, contrary to the Personal Digital Assistant (PDA) devices commonly used for picking, the *arDent Display* and the *ScanPad* are lightweight (both $60g$) and allow hands-free operation. Picking information

[1] The retrieval of items from a storage.

can be displayed on the *arDent Display*, such as object location or quantity, and the *ScanPad* allow to scan the bar code and encode values, e.g. the quantity taken. Finally, *Picking Pilot* allows to interface *Get Your Way*'s products with different ERPs, in order to retrieve the picking information, and return the picking result once finished.

Apart from continuing the development of their current products, *Get Your Way* is also designing new ones using feedback from organizations concerned by the picking issue. For instance, they have discovered that some operators do not appreciate using HMDs, especially those who have progressive lenses. To suit these operators, *Get Your Way* decided to develop the following products:

- A version of the *arDent Display* that is attached to the wrist, to be used with a connected *ScanPad*.
- A version of *ScanPad* which uses an e-ink touch screen
- A version of *ScanPad* which uses an AMOLED touch screen

These new products still retain the features that are dear to *Get Your Way*: they are lightweight and hands-free.

The main commercial offer is the *aRdent Pack*, described on figure 9.2. It contains a *ScanPad* and an *aRdent* display with its accessories. The *aRdent pack* is a Product as a Service (PaaS): maintenance, repairs, software updates and hardware upgrades are provided for a yearly fee. Similarly, *Picking Pilot* is provided as a Software as a Service (SaaS), giving access to software updates and tech support.

Figure 9.2: *arDent Pack*
Source: [77]

9.1.2 Target Markets

Get Your Way is mainly active in two markets, related to picking:

- The logistics sector. Companies with large warehouses usually already have a digital picking system, i.e. PDA devices connected to their company ERP. The *aRdent Pack* is a an alternative to PDA devices: the main selling points are the weight, the hand-free aspect and the ease of use. *Get Your Way* uses resellers to enter this market, which handle the integration of *aRdent Pack* with the client's ERP system. As logistics companies usually already have a software that links their PDA devices to their ERP system, *Picking Pilot* is not needed.

- The hospital pharmacy sector. Hospital pharmacies usually contain sizable storage areas, and have to perform daily pickings in order to dispatch medicine to patients. Apart from the convenient picking allow by the *aRdent Pack*, the traceability conferred by the *Picking Pilot* software is a huge selling point. Indeed, a recent Belgian law [78] forces hospitals to keep trace of the batch number of each administered drug. Since many Belgian hospitals still relied on paper to do medicine picking, this law incentives them to make the switch to digitization in order to fulfill this requirement in a convenient manner. This presents a great opportunity for *Get Your Way*.

  Regular pharmacies are not targeted because picking is less intensive for them, and they already fill the traceability requirement. Indeed, medicine packages are simply scanned during payment, so linking the batch numbers to the clients is easy.

Get Your Way actively prospects for new clients: they regularly send delegates to logistics and pharmacist congresses, in Belgium and abroad, to organize demonstrations of their products.

The company is also considering about targeting the industrial sector using the *arDent Display* as a solution to the procedure issue, i.e. follow a long procedure which can not be remembered, which implies that it must be written somewhere accessible to the operator. For now however, *Get Your Way* currently focuses on hospitals.

9.2 Internship

The internship including report redaction lasted 95 days, 70 of which spent inside the company office. The internship focused entirely on Research and Development (R&D), with the intent of integrating the work performed into *Get Your Way*'s flagship product: the *arDent Display*. *Get Your Way* organized weekly meetings on Mondays where employees and interns would summarize the work performed the week prior, and their objectives for the current week. This allowed to obtain an overview of the issues faced by the company, such as: R&D, patenting, marketing, communication, and the whole commercial life cycle (i.e. from prospecting to deployment).

This thesis is particularly valuable to the company, as they do not possess in-house expertise in the implemented standards. Indeed, the USB software module used in the *arDent Display* was implemented by external contractors, and the UVC standard was yet to be studied for implementation. Nonetheless, the internship supervisor and R&D team provided helpful advice and insights about embedded software design in general, and the used microcontroller.

Overall, this internship proved to be a valuable learning experience in a professional setting.

Bibliography

[1] URL: https://www.getyourway.be.

[2] Get Your Way. "aRdent Hands-free solutions - Products Catalog".

[3] URL: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/.

[4] URL: https://intradefairs.com/digital-visual-interface.

[5] URL: https://www.hdmi.org/spec/hdmi2_1.

[6] URL: https://en.wikipedia.org/wiki/IEEE_1394.

[7] EMVA. "GenICam GenTL Standard". Version 1.6.

[8] URL: https://en.wikipedia.org/wiki/PCI_Express#.

[9] URL: https://pcisig.com/pci-express-6.0-specification.

[10] URL: https://www.technexion.com/resources/mipi-cameras-definition-types-and-applications/.

[11] URL: https://www.mipi.org/specifications/csi-2.

[12] URL: https://www.mipi.org/specifications/camera-command-set.

[13] Dr. Thomas Rademacher and Frank Karstens. "The Significance of the MIPI CSI-2 Interface in Embedded Vision Applications". White paper, Basler, 2017.

[14] Arasan Chip Systems Inc. "C-PHY vs D-PHY - Choosing the right signal interface for MIPI Camera and Imaging". White paper, 2016.

[15] Edo Cohen, Raj Kumar Nagpal, George Wiley and Kirill Dimitrov. "What's New and Coming Up on the MIPI PHY Roadmap". URL: https://www.mipi.org/knowledge-library/conference-presentations/mipi-devcon-2022-phy-roadmap.

[16] STMicroelectronics. "MIPI A-PHY EOS protection in automotive applications". Application Note AN5891.

[17] URL: https://www.spinelelectronics.com/what-is-the-difference-between-dvp-and-mipi-cameras/.

[18] URL: https://www.sct.com.tw/articles/what-is-sdi-understanding-sdi-video-standards.

[19] URL: https://www.adimec.com/pros-and-cons-hd-sdi-3g-sdi-for-global-security/.

[20] AIA. "Specifications of the Camera Link Interface Standard for Digital Cameras and Frame Grabbers". Version 2.0.

[21] URL: https://www.automate.org/vision/vision-standards/vision-standards-camera-link-hs.

[22] Allied Vision. "GenICam-yet another standard ?". White Paper version 1.

[23] URL: https://www.coaxpress.com/.

[24] URL: https://www.e-consystems.com/blog/camera/technology/what-is-gmsl-technology-and-how-does-it-work/.

[25] URL: https://www.theimagingsource.com/en-us/embedded/fpd-link-iii/.

[26] Casey McCrea. "What is FPD-Link?".

[27] URL: https://tripplite.eaton.com/products/ethernet-cable-types.

[28] URL: https://www.edge-core.com/blogs/towards-800g-and-1600g-ethernet/.

[29] URL: https://antmedia.io/ip-camera-streaming-guide-how-to-setup-an-ip-camera/.

[30] Basler. "The Elements of GigE Vision". White paper.

[31] URL: https://www.automate.org/vision/vision-standards/vision-standards-gige-vision.

[32] URL: https://va-imaging.com/blogs/machine-vision-knowledge-center/how-to-select-a-machine-vision-camera-interface-usb3-gige-5gige-10gige-vision.

[33] URL: https://lumenci.com/blogs/thunderbolt-interface/.

[34] Sebastian Guenther, Mar. "How to overcome cable length limitations in MIPI CSI 2 based systems". URL: https://www.mipi.org/knowledge-library/conference-presentations/mipi-sessions-embedded-world-2023.

[35] Cypress. "AN57294 - USB 101: An Introduction to Universal Serial Bus 2.0". Document 001-57294.

[36] URL: https://microchip.my.site.com/s/article/What-is-the-maximum-theoretical-bandwidth-of-a-USB2-0-device.

[37] Donovan (Don) Anderson. "Introduction to USB 3.0". White paper, Mindshare.

[38] URL: https://www.everythingusb.com/speed.html.

[39] Extron Electronics. "Understanding USB in Professional AV Environments". White Paper, Revision 1.0, 2017.

[40] USB-IF. "Universal Serial Bus 4 (USB4®) Specification". Version 2.0.

[41] URL: https://www.usb.org/defined-class-codes.

[42] USB-IF. "Universal Serial Bus Device Class Definition for Video Devices". Revision 1.5.

[43] URL: https://www.pleora.com/about-us/standards-leadership/usb3-vision/.

[44] URL: https://www.totalphase.com/blog/2024/12/what-is-usb-otg-and-how-does-it-work/.

[45] USB-IF. "On-The-Go and Embedded Host Supplement to the USB Revision 3.0 Specification". Revision 1.1.

[46] USB-IF. "Universal Serial Bus Type-C Cable and Connector Specification". Release 2.4.

[47] USB-IF. "Universal Serial Bus Power Delivery Specification". Version 1.1, Revision 3.2.

[48] VESA. "VESA Display Stream Compression". White Paper ETP200, 2014.

[49] URL: https://www.supertekmodule.com/choosing-the-right-video-compression-format-a-comparison-of-h-264-h-265-and-mjpeg/.

[50] Dr.Sabeenian R.S. "Comparison of Video Compression Standards". *International Journal of Computer and Electrical Engineering*, 5(6):549–554, 2013.

[51] URL: https://www.mpeg.org/.

[52] ISO/IEC JTC 1 and ITU-T. "Information technology - Coding of audio-visual objects - Part 2: Visual". Standard ISO/IEC 14496-2.

[53] Axis Communications. "H.264 video compression standard". White Paper 31669/EN/R1/0803, 2008.

[54] URL: https://stl.tech/blog/optical-interconnect-vs-electrical-interconnect-all-you-need-to-know/.

[55] URL: https://www.intopix.com/blogs/post/Revolutionizing-SDI-Video-Transmission-JPEG-XS-Compression-with-intoPIX-SDI-Mapping-Techniques.

[56] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips. "Universal Serial Bus Specification". Revision 2.0.

[57] USB-IF. "Universal Serial Bus Device Class Definition for Video Devices". Revision 1.1.

[58] USB-IF. "Universal Serial Bus Device Class Definition for Video Devices: Motion-JPEG Payload". Revision 1.1.

[59] Intel Corporation. "Enhanced Host Controller Interface Specification for Universal Serial Bus". Revision 1.0.

[60] Renesas. "RZ/A2M SUB Board RTK79210XXB00000BE User's Manual". Application Note Rev.2.0.

[61] Renesas. "RZ/A2M CPU Board RTK7921053C00000BE User's Manual". Application Note Rev.2.0.

[62] Renesas. "RZ/A2M Group User's Manual: Hardware". Rev. 4.00.

[63] URL: https://developer.arm.com/documentation/ddi0388/h/Memory-Management-Unit/About-the-MMU/Memory-Management-Unit?lang=en.

[64] Renesas. "RZ/A2M Group USB Basic Host Driver". Application Note R01AN4715EJ0150, Rev.1.50.

[65] URL: https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/03-Direct-to-task-notifications/01-Task-notifications.

[66] URL: https://community.renesas.com/rz/f/rz-forum/53724/usb-isochronous-request-with-r_usbh0-basic-driver.

[67] Renesas. "RZ/A2M Group USB Host Mass Storage Class Driver". Application Note R01AN4714EJ0151, Rev.1.51.

[68] Renesas. "RZ/A2M Group JPEG Codec Unit "JCU" Driver Example". Application Note R01AN4456EJ0110, Rev.1.10.

[69] Renesas. "RZ/A2M Group JPEG Codec Unit "JCU" Sample Application". Application Note R01AN4492EJ0113, Rev. 1.13.

[70] Renesas. "RZ/A2M Group Camera and Display sample". Application Note R01AN4477EJ0114 Rev.1.14.

[71] URL: https://www.collabora.com/news-and-blog/blog/2016/02/16/a-programmers-view-on-digital-images-the-essentials/.

[72] URL: https://en.wikipedia.org/wiki/JPEG/.

[73] MALAY Thibault. Connecteur.SchDoc.

[74] Texas Instruments. "TUSB320 USB Type-Câ¢ Configuration Channel Logic and Port Control". Datasheet SLLSEN9F.

[75] URL: https://www.renesas.com/en/key-technologies/artificial-intelligence/drp.

[76] Techno Mathematical Co. "TMC DRP compliant H.264 Codec". Product brief.

[77] Get Your Way. "Guide d'utilisation aRdent - Picking pilot".

[78] Agence fédérale des médicaments et des produits de santé'. "Arrêté royal du 30 Septembre 2020, Art. 32". URL: https://www.ejustice.just.fgov.be/eli/arrete/2020/09/30/2020031582/moniteur.