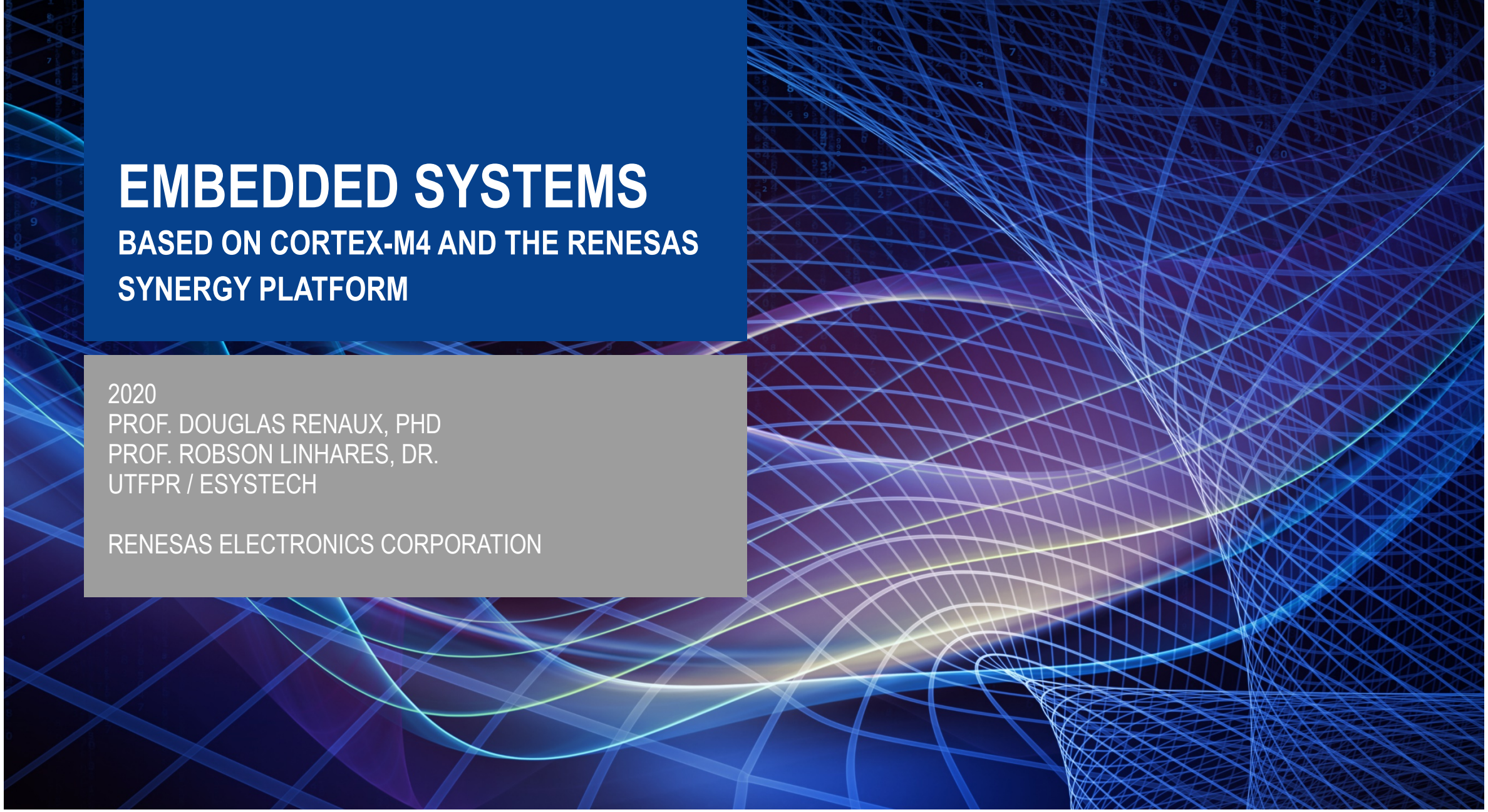


EMBEDDED SYSTEMS

BASED ON CORTEX-M4 AND THE RENESAS SYNERGY PLATFORM

2020
PROF. DOUGLAS RENAUX, PHD
PROF. ROBSON LINHARES, DR.
UTFPR / ESYSTECH

RENESAS ELECTRONICS CORPORATION

10 – USB

- Introduction
- Block Diagram
- Registers
- SW Stack

10.1 – INTRODUCTION

USB is an acronym for **Universal Serial Bus**. It has been proposed by a consortium of companies, such as Microsoft, Intel, IBM, Compaq and NEC and is designed to support a wide range of applications that require communication with distinct characteristics (real-time, high or low bandwidth, with or without message delivery guarantee etc.).

Current specification is 3.2 (Sep, 2017).

10.1 – INTRODUCTION

Examples of devices that make use of USB:

- Printers
- Cameras → interface for photo and video upload
- Smartphones → interface for battery charging and file transfer
- Human Interface Devices → keyboard, mouse etc.
- Development electronic boards → debug interface (JTAG emulation)
- Game joysticks
- …

10.1 – INTRODUCTION

Characteristics of USB:

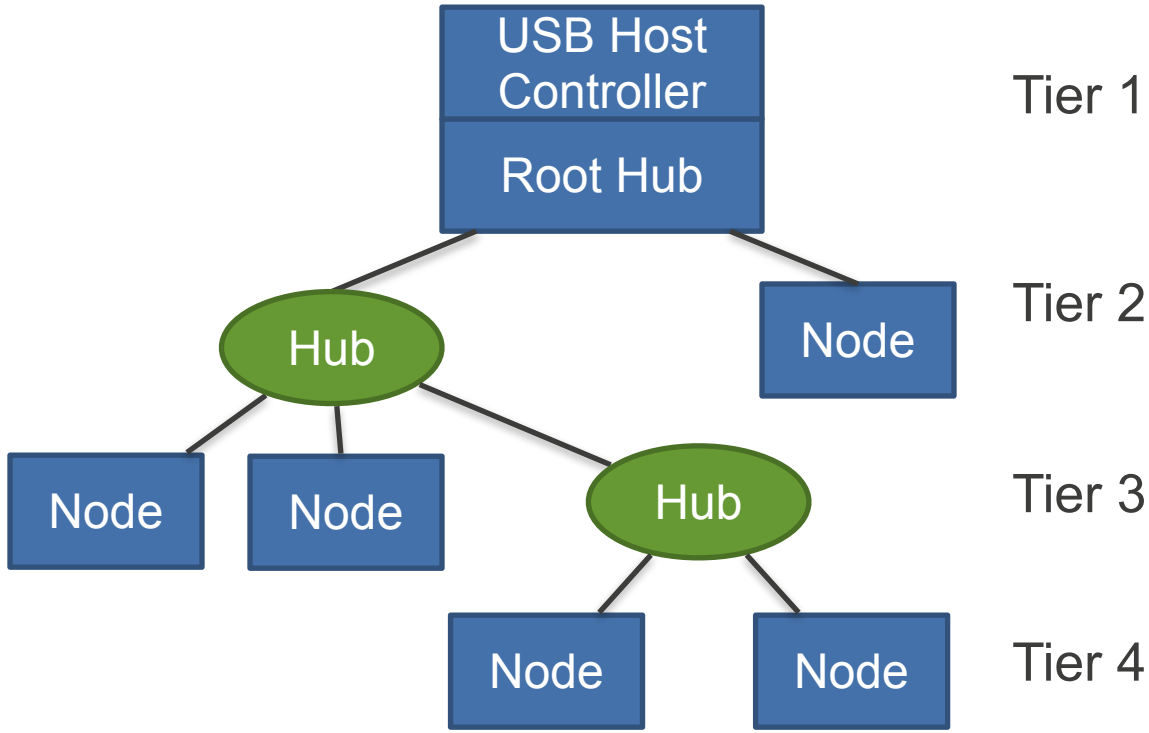
- Four (or five)-wire serial bus with single master (host) and up to 127 slaves (devices);
  - Exception → USB On-The-Go (OTG) → allows negotiation between two devices (point to point) to be a temporary host;
  - Example of OTG use: a camera device connected to a printer device to print photos;
- Defines low speed (1.5 Mbps), full speed (12 Mbps) and high speed (up to 5 Gbps at version 3.0) bandwidth
- Rem: USB 3.0 uses a 9-pin connector (USB-A 3.0 connector) or a 24-pin USB-C connector.

10.1 – INTRODUCTION

- Four types of data transfers → meet the requirements of different communication types
- Device class identification by the host via enumeration protocol → allows plug-and-play and hot swap capabilities, as well as instantiation of the proper class driver software by the host
- Bus power capabilities → some USB devices do not need an extra power source
  - USB host is able to detect overcurrent conditions, so that power can be removed from the device causing the problem without affecting the other devices already connected

USB TOPOLOGY

- USB **Host** Controller is the master and generates transactions (via Root Hub).
- Each Hub is physically connected (by wire) to a Node or another Hub.
- Nodes are slaves which perform the functions → also known as **Devices**.
- Each level defines a tier → maximum of 7 as USB 2.0 Specification.
- Nodes can be inserted or removed when necessary → upon insertion, the enumeration process is executed to identify the device class and configure it.



source: Authors

USB PHYSICAL INTERFACE

USB 1.1 and 2.0 → 4 shielded wires.

- 2 wires for data → differential
- 2 wires for power (5 Vdc and GND)

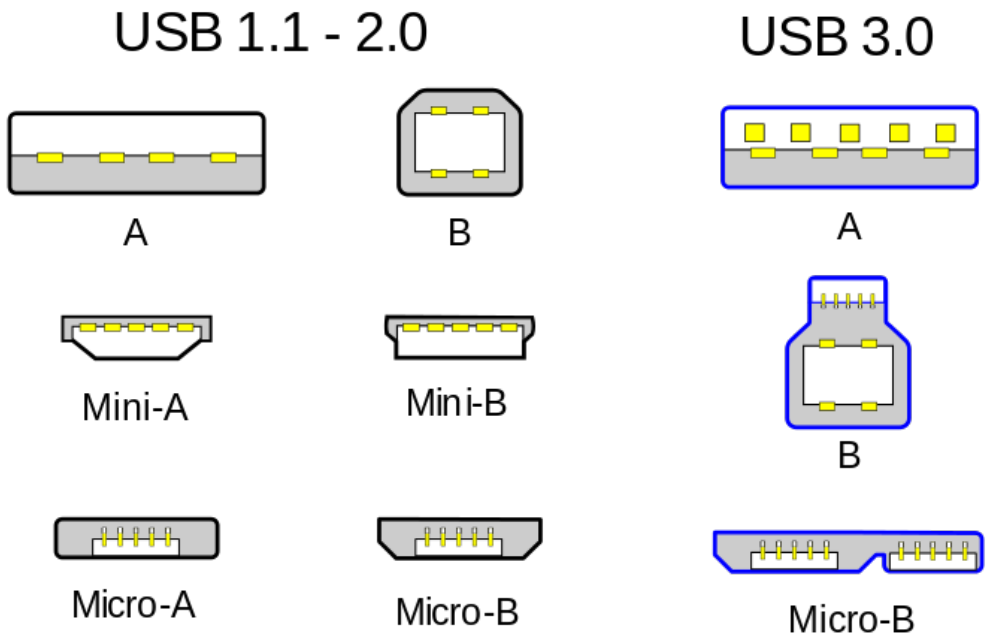
Type A → host.

Type B → device.

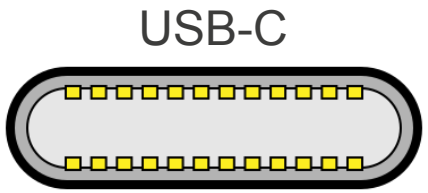
Mini and Micro variations use the same electrical interface in smaller form factors.

OTG → extra pin to identify the role of the device (A or B). The receptacle is called Micro-AB and accepts both Micro-A and Micro-B connectors.

USB 3.0 → 5 extra wires.



Source: By Milos.bmx (Own work) [CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons https://commons.wikimedia.org/wiki/File%3AUSB3.0_connectors.svg



CC0
https://commons.wikimedia.org/wiki/File:USB_Type-C_icon.svg

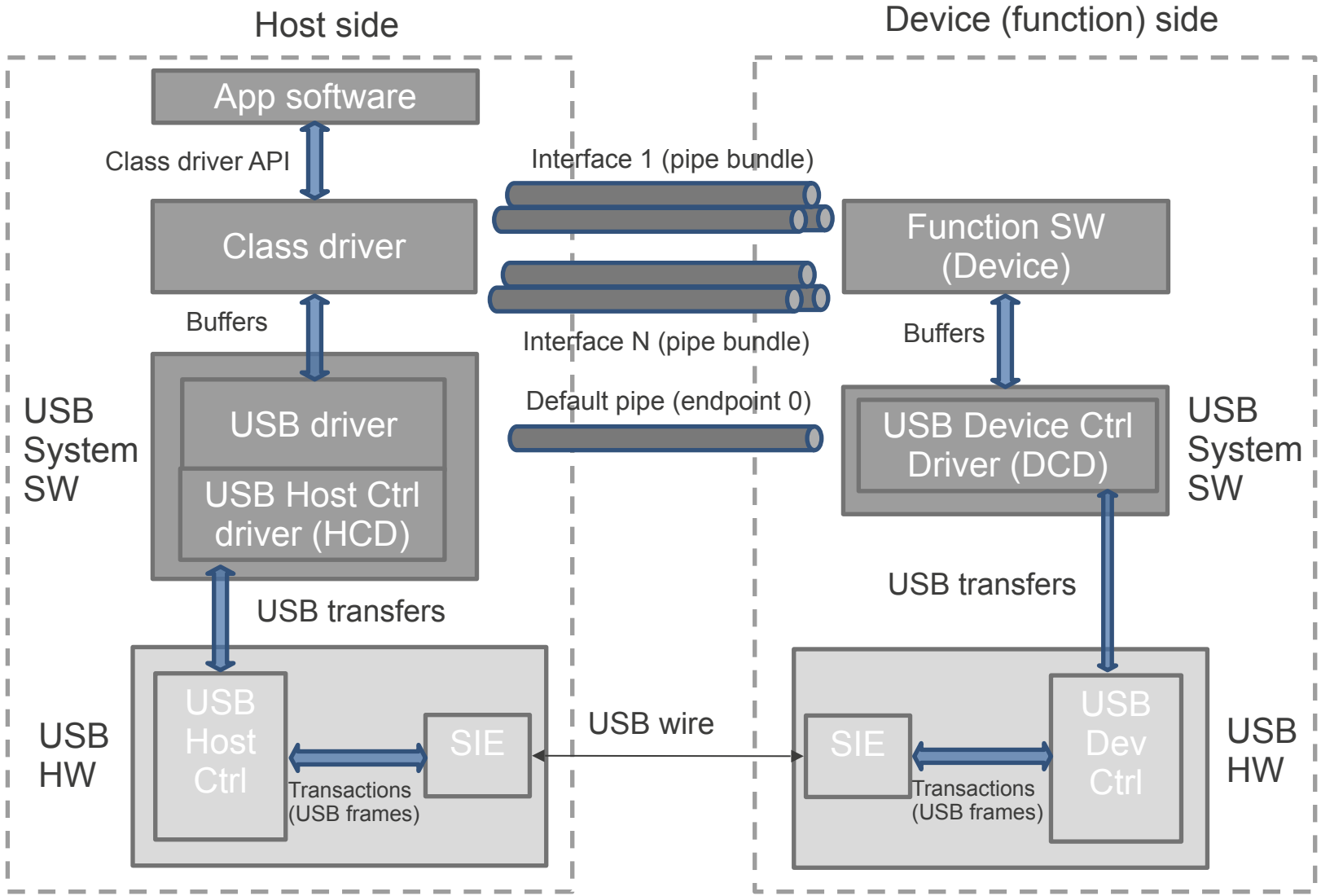
USB LOGICAL VIEW

Communication flows are performed via **pipes** → composed of **endpoints** → unidirectional data paths.

Endpoint / pipe bundles form an **interface** → a view to the function / device behavior as it is exposed to the host.

The host side instantiates a **class driver** during device enumeration → manages the interfaces and provides an API to the app level.

Default pipe is bidirectional (endpoint 0 in both directions) and is used for device configuration.



source: Authors

USB HOST CONTROLLER

- The USB Host Controller hardware layer offers an HCI (Host Controller Interface) to the Host Controller Driver in software → standardizes the register access and allows interoperability between the host OS and different hardware implementations.
- Some HCI standards have been historically defined:
  - **OHCI** (Open Host Controller Interface) → defined for USB 1.1, manages the USB bus mainly in hardware (internal FIFO descriptors management)
  - **UHCI** (Universal Host Controller Interface) → proprietary interface by Intel, defined for USB 1.1, manages most of the USB bus operation in software (HCD level)

USB HOST CONTROLLER

- **EHCI** (Enhanced Host Controller Interface) → defined for USB 2.0, manages high-speed communication on a USB bus. EHCI controllers have been usually implemented in PC motherboards in conjunction with UHCI or OHCI drivers (that managed the low and full-speed devices).
- **xHCI** (Extensible Host Controller Interface) → defined for USB 3.0, manages all the USB bus speeds. It is meant to replace the previous UHCI/OHCI/EHCI standards.

USB PACKETS

USB Transfers are performed by a sequence of transactions, which are composed of:

- A **Token** packet, carrying addressing, direction and packet type information (IN, OUT or SETUP). The token can be of PID type **Start-Of-Frame (SOF)**, issued every 1 ms (full-speed) or 125 us (high-speed) and used for synchronization.
  - Frame → interval during which a sequence of transactions is performed for the endpoints controlled by the host..

Field	PID	ADDR	ENDP	CRC5
Bits	8	7	4	5
Desc	Type of packet	Device address	Endpoint address	CRC of ADDR and ENDP

Token packet

Field	PID	Frame number	CRC5
Bits	8	11	5
Desc	Type of packet	Current frame	CRC of Frame Number

SOF packet

USB PACKETS

- A **Data** packet, carrying the effective data being transferred. Data packets are issued either by the host or the device, depending on the endpoint direction (identified by the previous Token packet). PID indicates type DATA0 or DATA1 (toggling for full-speed transfers) or DATA2 (high-speed transfers).
- A **Handshake** packet, used to report the status of a data transaction. Handshake packets are issued by the receiver of the Data packet. PID indicates ACK, NACK, a halt condition (STALL) or no response yet (NYET).

Field	PID	DATA	CRC16
Bits	8	0-8192	16
Desc	Type of packet	Data	CRC of DATA

Data packet

Field	PID
Bits	8
Desc	Type of packet

Handshake packet

USB TRANSFERS

USB defines four types of data transfers:

- **Control** → control commands to configure device, delivery guaranteed, low bandwidth required.
  - Control transfers are performed in three stages:
    - A *Setup* stage, starting with a token packet of type SETUP and a data packet containing a USB device request → see following slides.
    - An optional *Data* stage, starting with a token packet of type IN or OUT (depending on direction) and a data packet containing the data pertaining to the USB device request.
    - A *Status* stage, starting with a token packet of type IN or OUT (inverse of Data direction) and containing request status information.

USB TRANSFERS

- **Bulk** → large amounts of data, non real-time, delivery guaranteed, variable use of bandwidth → used for reliable data transfers, such as mass storage data.
  - Token packets for bulk transfers are of type IN or OUT, depending on transfer direction.
- **Interrupt** → real-time, small and periodic amounts of data → used for event notification (e.g. key typed on a keyboard).
  - Token packets for interrupt transfers are of type IN or OUT, depending on transfer direction.

USB TRANSFERS

- **Isochronous** → large amounts of data, delivery not guaranteed, steady rate of transmission and reception, bandwidth depending on sampling characteristics → used for streaming data, such as voice or video.
  - Token packets for isochronous transfers are of type IN or OUT, depending on transfer direction.
  - Isochronous transfers do not use Handshake packets.

Individual endpoints are configured to a specific type of data transfer, depending on the class driver loaded by the host during the enumeration process → see following slides.

USB TRANSFERS SCHEDULING

Rules for transfer scheduling:

- Periodic transfers (isochronous and interrupt) → limited to 90% of the bandwidth of a frame.
- Control → use as much as necessary of the remaining 10% (plus the remaining amount in the 90% of the bandwidth that is not used for periodic transfers).
- Bulk → use the bandwidth that is left.

USB ENUMERATION

The USB enumeration protocol is executed whenever a new device is inserted into the bus. This protocol comprises the following steps:

- USB root hub detects when a device is connected (D- or D+ are pulled up with resistors).
- USB host powers and resets the device.
- USB host issues device requests through the Default Control Pipe (default address 0) to get the Device Descriptor → see following slides.
- USB host assigns a unique address to the device.
- USB host issues device requests through the Default Control Pipe to get the Configuration Descriptors → see following slides.
- USB host enables a valid configuration → all corresponding interfaces and endpoints are configured, and the device may draw the current described in the descriptor for the selected configuration.

USB ENUMERATION

Some steps of the enumeration protocol require issuing USB device requests to the device being enumerated.

The USB device requests are issued during the Setup stage of a Control transfer. A device request is 8 bytes long and contains the following fields:

- *bmRequestType* (1 byte) → request direction, type (standard, class, vendor, reserved) and recipient (device, interface, endpoint, other).
- *bRequest* (1 byte) → specific request (set address, get and set configuration, get and set descriptor, get and set interface etc.).
- *wValue* (2 bytes), *wIndex* (2 bytes) → value and index that depend on request.
- *wLength* (2 bytes) → number of bytes to transfer if there is a data stage.

Refer to USB 2.0 Specification, Section 9.3 for more detailed information.

USB ENUMERATION

Descriptors sent by the device during enumeration process (in response to GET_DESCRIPTOR requests):

- **Device descriptor** → defines the device class, device subclass, device protocol, max packet size for default endpoint, vendor, product, release number, indices for manufacturer, product and serial number strings, and the number of configurations.

Device Descriptor

Field	Size (bytes)	Descr
bLength	1	Size of descriptor
bDescriptorType	1	DEVICE descriptor type
bcdUSB	2	USB Spec Relase Number in BCD
bDeviceClass	1	Class code
bDeviceSubClass	1	Subclass code
bDeviceProtocol	1	Protocol code
bMaxPacketSize0	1	Max packet size for endp 0
idVendor	2	Vendor ID
idProduct	2	Product ID
bcdDevice	2	Device release number in BCD
iManufacturer	1	Index of string desc for manufacturer
iProduct	1	Index of string desc for product
iSerialNumber	1	Index of string desc for serial
bNumConfigurations	1	Number of possible configurations

Source: USB 2.0 Specification, Table 9-8

USB ENUMERATION

- **Configuration descriptor** → defines the number of interfaces for this configuration, the configuration value and an index for this configuration's string, if the device is self-powered when running that configuration and the max power consumption (in case it is bus powered).
  - A GET_DESCRIPTOR request to a Configuration Descriptor returns also the Interface and Endpoint descriptors pertaining to the given Configuration, in sequential order → see next slides.

Configuration Descriptor

Field	Size (bytes)	Descr
bLength	1	Size of descriptor
bDescriptorType	1	CONFIGURATION descriptor type
wTotalLength	2	Total length of configuration data (includes Interface and Endpoint descriptor sizes)
bNumInterfaces	1	Number of interfaces
bConfigurationValue	1	Configuration ID
iConfiguration	1	Index of string desc for this config
bmAttributes	1	Configuration characteristics
bMaxPower	1	Max power consumption in mA when operating on this configuration

Source: USB 2.0 Specification, Table 9-10

USB ENUMERATION

- **Interface descriptor** → defines the interface number, the number of endpoints, the interface class, subclass and protocol, and an index to a string describing this interface.

Interface Descriptor

Field	Size (bytes)	Descr
bLength	1	Size of descriptor
bDescriptorType	1	INTERFACE descriptor type
bInterfaceNumber	1	Zero-based number of this interface
bAlternateSetting	1	Value to select this alternate setting
bNumEndpoints	1	Number of endpoints used by this interface
bInterfaceClass	1	Class code for this interface
bInterfaceSubClass	1	Subclass code for this interface
bInterfaceProtocol	1	Protocol code for this interface
iInterface	1	Index of string desc for this interface

Source: USB 2.0 Specification, Table 9-12

USB ENUMERATION

- **Endpoint descriptor** → defines the endpoint address and direction, the endpoint type (control, isochronous, bulk or interrupt), the max packet size and the polling interval for periodic endpoints (isochronous and interrupt).

- Refer to USB 2.0 Specification, Section 9.6 for more detailed information.

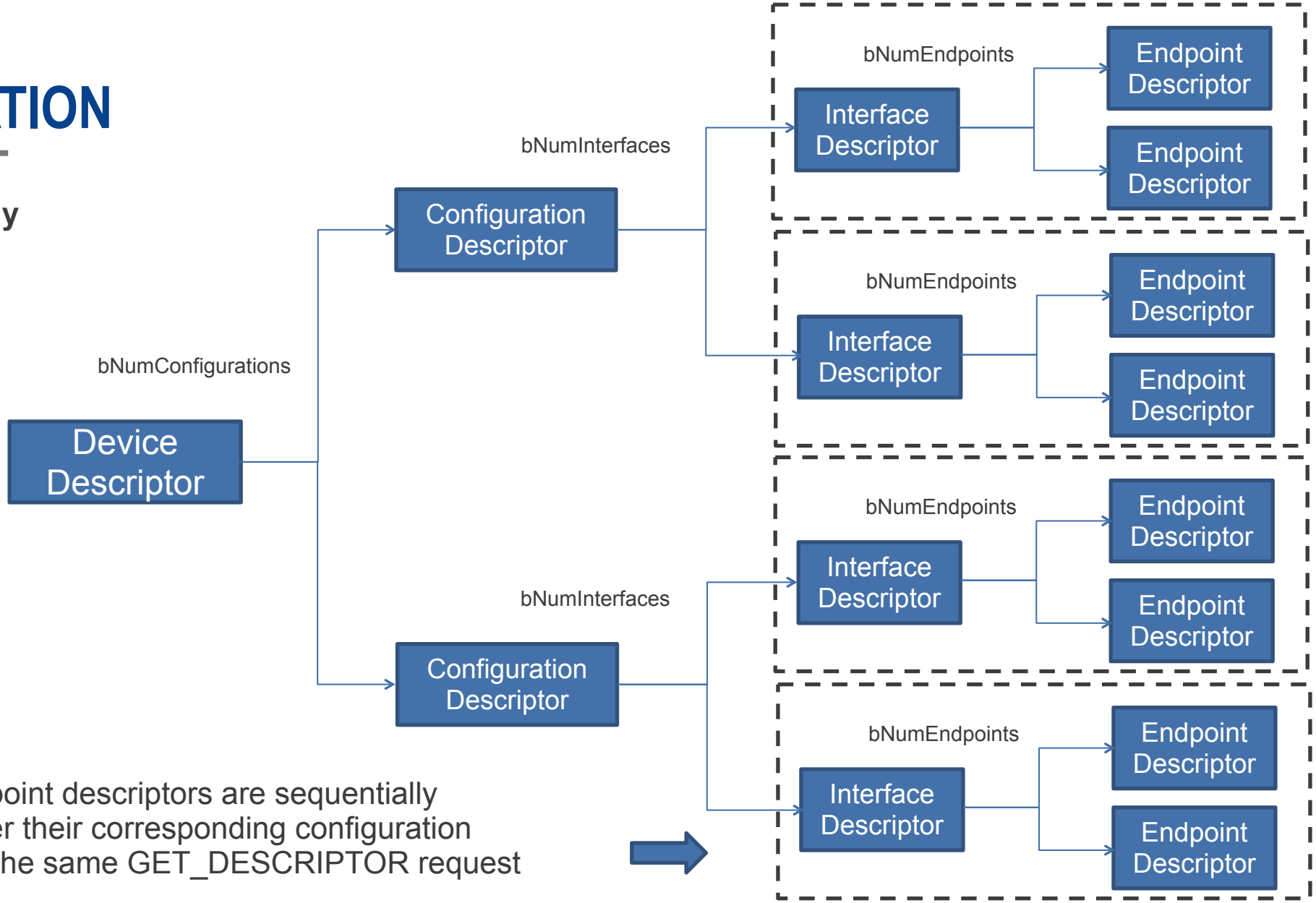
Endpoint Descriptor

Field	Size (bytes)	Descr
bLength	1	Size of descriptor
bDescriptorType	1	ENDPOINT descriptor type
bEndpointAddress	1	Address for this endpoint
bmAttributes	1	Endpoint attributes (type etc.)
wMaxPacketSize	2	Max packet size for this endpoint
bInterval	1	Polling interval in frames

Source: USB 2.0 Specification, Table 9-13

USB ENUMERATION

USB Descriptor hierarchy



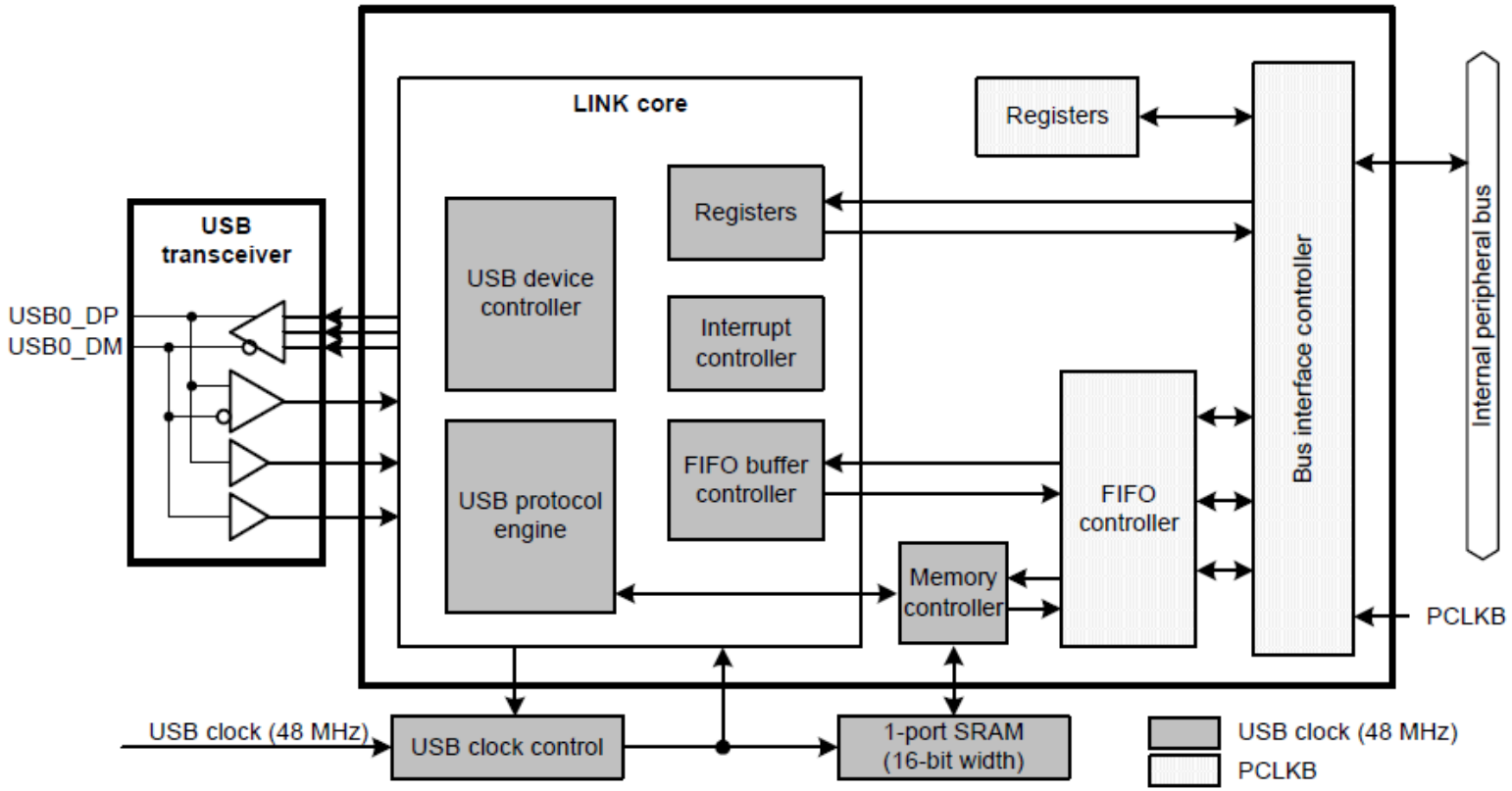
Interface and endpoint descriptors are sequentially retrieved, right after their corresponding configuration descriptor, during the same GET_DESCRIPTOR request

Source: Authors

10.2 – BLOCK DIAGRAM – CASE STUDY

The R7FS7G27H3A01CFC Renesas ARM Cortex-M4 MCU implements two USB modules:

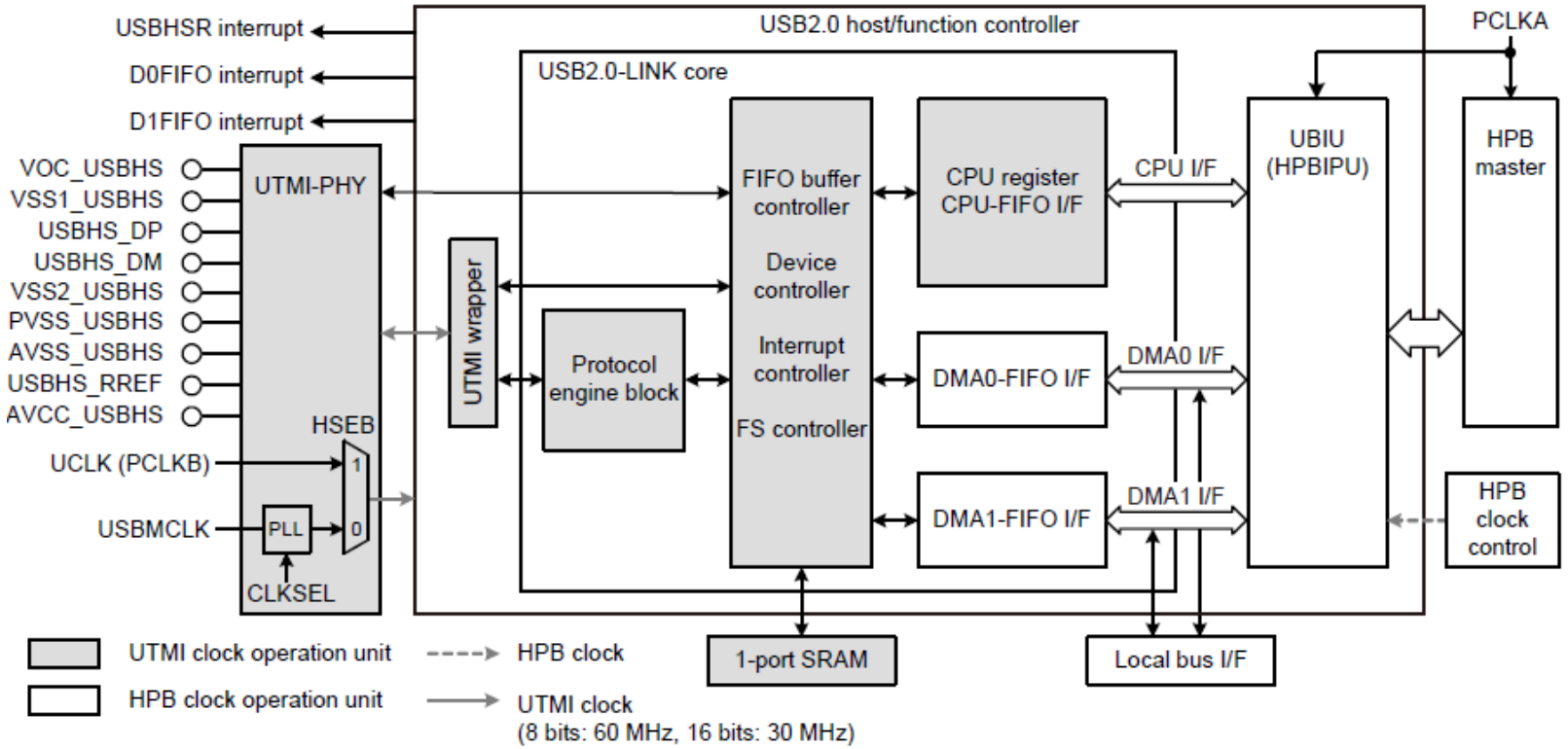
- **USB 2.0 FS** → operates only on low and full speed modes. Based on registers and a FIFO controller to manage buffers to be received / transmitted.



Source: Renesas Synergy MCUs User's Manual: Hardware

10.2 – BLOCK DIAGRAM – CASE STUDY

- USB 2.0 HS → operates in high speed mode (480 Mbps). Uses DMA FIFOs to maximize memory transfer speed.



Source: Renesas Synergy MCUs User's Manual: Hardware

10.3 – REGISTERS – CASE STUDY

Implementation for the USB 2.0 FS Module of the R7FS7G27H3A01CFC Renesas ARM Cortex-M4 MCU:

- SYSCFG → enabling/disabling USB, pull up / pull down resistor config
- SYSSTS0 → line status, overcurrent status (from an external overcurrent detector), status bits for entering / exiting the "suspended" mode
- DVSTCTR0 → connection status (reset, low-speed or full-speed), wakeup detection, enable / resume / reset control
- CFIFO, D0FIFO and D1FIFO → read/write from/to FIFOs associated to control pipe and to other communication pipes
- CFIFOSEL → configure control pipe and associate to CFIFO
- D0FIFOSEL, D1FIFOSEL → associate pipes to D0FIFO and D1FIFO, configure DMA

10.3 – REGISTERS – CASE STUDY

- CFIFOCTR, D0FIFOCTR, D1FICOCTR → received data length, status of FIFO read
- INTENB0, INTENB1 → enable / disable USB interrupts
- BRDYENB → enable / disable BRDY (data transfer successful) interrupt for each USB pipe
- NRDYENB → enable / disable NRDY (data transfer not successful) interrupt for each USB pipe
- BEMPENB → enable / disable BEMP (buffer empty or incorrect packet size) interrupt for each USB pipe
- SOFCFG → configuration for SOF (start-of-frame) and frame timing (LS and FS)
- INTSTS0, INTSTS1 → status of several interrupt sources (SOF, resume, BRDY, NRDY, overcurrent, disconnection etc.)

10.3 – REGISTERS – CASE STUDY

- BRDYSTS → status of BRDY interrupt for each USB pipe
- NRDYSTS → status of NRDY interrupt for each USB pipe
- BEMPSTS → status of BEMP interrupt for each USB pipe
- FRNUM → frame number, status of CRC error and overrun/underrun in isochronous transfers
- DVCHGR → used when device recovers from deep software standby mode due to USB events
- USBADDR → USB device address, configuration for recovery from deep software standby mode
- USBREQ → fields of setup requests used for control transfers.
- USBVAL → stores the wValue field of setup transactions (received and for transmitting)

10.3 – REGISTERS – CASE STUDY

- USBLENG → stores the wLengths field of setup transactions (received and for transmitting)
- DCPCFG → enabling and direction of the Default Control Pipe
- DCPMAXP → maximum packet size for the Default Control Pipe
- DCPCTR → controls transfers for the Default Control Pipe
- PIPESEL → select pipe to be configured by PIPECFG, PIPEMAXP etc.
- PIPECFG → configures selected pipe (endp number, direction, transfer type etc.)
- PIPEMAXP → configures maximum packet size for selected pipe
- PIPEPERI → configures error detection interval for isochronous pipes
- PIPECTR[1..9] → controls transfers for the corresponding pipe
- PIPETRE[1..5] → enables / disables transaction counter

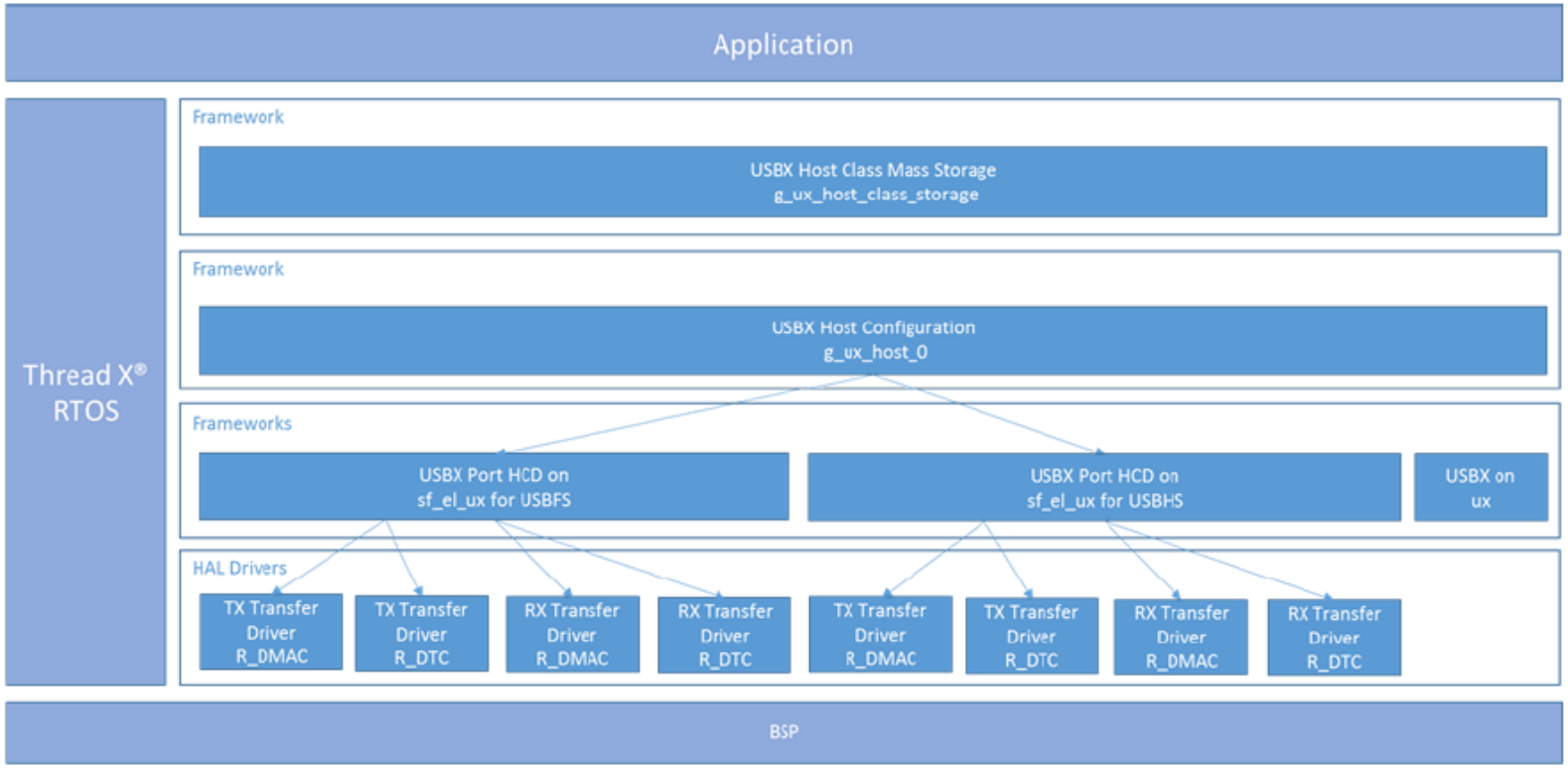
10.3 – REGISTERS – CASE STUDY

- PIPETRN[1..5] → transaction counters for the corresponding pipes
- DEVADD[0..5] → configures the transfer speed for the device to which the corresponding pipe is communicating
- PHYSLEW → adjust the physical driver to host or function operation
- DPUSR0R → configures pull-up / pull-down resistors, reads status of overcurrent and VBUS inputs
- DPUSR1R → configures and reads status concerning deep software standby mode
- USBMC → enables / disables battery charging mode and regulator circuit
- USBBCCTRL0 → configures parameters for battery charging mode

10.4 – SOFTWARE STACK – CASE STUDY

- Example of USB Host stack for Renesas microcontroller hardware (part of SSP – Synergy Sofware Package):
- Uses a Mass Storage Module on top as class driver.
- Uses Thread X RTOS to manage the threads concerning USB components.

- (https://www.renesas.com/en-us/software/D6001255.html)



Source: Renesas Synergy USBX Host Class Mass Storage Module Guide
r11an0173eu0100-synergy-ux-host-class-mass-storage-mod-guide

10.4 – SOFTWARE STACK – CASE STUDY

- The application for the shown example uses the top-level API provided by the USBX Host Class Mass Storage component:
- This component instantiates a file manager (FileX) when a mass storage device (e. g. an USB memory) is inserted.
- The application uses the API provided by FileX to access the mass storage device contents → file open, close, read, write etc.

```
UINT    fx_file_allocate(FX_FILE *file_ptr, ULONG size);
UINT    fx_file_attributes_read(FX_MEDIA *media_ptr, CHAR
            *file_name, UINT *attributes_ptr);
UINT    fx_file_attributes_set(FX_MEDIA *media_ptr, CHAR
            *file_name, UINT attributes);
UINT    fx_file_best_effort_allocate(FX_FILE *file_ptr, ULONG
            size, ULONG *actual_size_allocated);
UINT    fx_file_close(FX_FILE *file_ptr);
UINT    fx_file_create(FX_MEDIA *media_ptr, CHAR *file_name);
UINT    fx_file_date_time_set(FX_MEDIA *media_ptr, CHAR *file_name,
            UINT year, UINT month, UINT day, UINT hour, UINT minute, UINT second);
UINT    fx_file_delete(FX_MEDIA *media_ptr, CHAR *file_name);
UINT    fx_file_open(FX_MEDIA *media_ptr, FX_FILE *file_ptr,
            CHAR *file_name, UINT open_type);
UINT    fx_file_read(FX_FILE *file_ptr, VOID *buffer_ptr, ULONG
            request_size, ULONG *actual_size);
UINT    fx_file_relative_seek(FX_FILE *file_ptr, ULONG byte_offset, UINT seek_from);
UINT    fx_file_rename(FX_MEDIA *media_ptr, CHAR *old_file_name, CHAR *new_file_name);
UINT    fx_file_seek(FX_FILE *file_ptr, ULONG byte_offset);
UINT    fx_file_truncate(FX_FILE *file_ptr, ULONG size);
UINT    fx_file_truncate_release(FX_FILE *file_ptr, ULONG size);
UINT    fx_file_write(FX_FILE *file_ptr, VOID *buffer_ptr, ULONG size);
```

Basic API functions

Source: FileX Services (https://rtos.com/wp-content/uploads/2017/10/EL-filex-programmers-guide.pdf)

Renesas.com