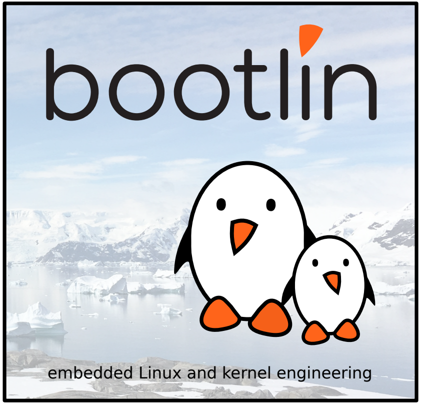
A tour of USB Device Controller (UDC) in Linux

Hervé Codina
herve.codina@bootlin.com



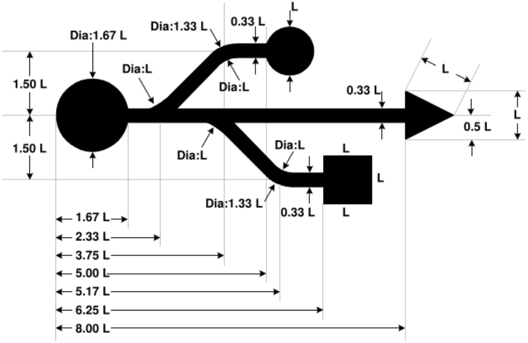
Hervé Codina

- Embedded Linux engineer at Bootlin
  - Embedded Linux **expertise**
  - **Development**, consulting and training
  - Contributor to the **Renesas RZ/N1 USBF** UDC driver in Linux
  - Strong open-source focus
- Open-source contributor
- Living in **Toulouse**, France

USB2.0 standard

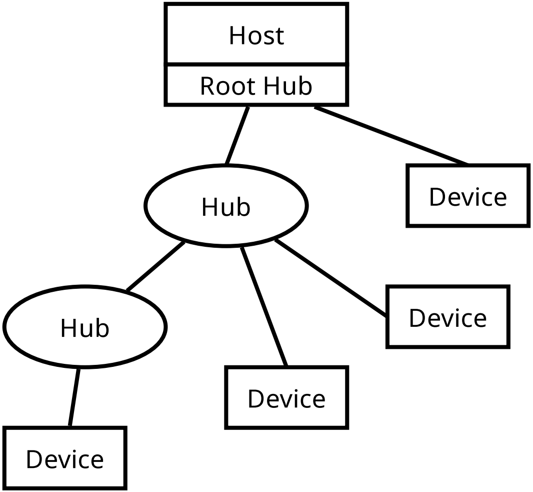
USB2.0 standard

- https://www.usb.org/document-library/usb-20-specification
- Released on April 27, 2000
- Defines the mechanical part, the electrical and communication protocol
- Publicly available
- Supports
  - High-speed (480 Mb/s)
  - Full-speed (12 Mb/s)
  - Low-speed (1.5 Mb/s)



Extracted from the USB2.0 standard

USB Bus



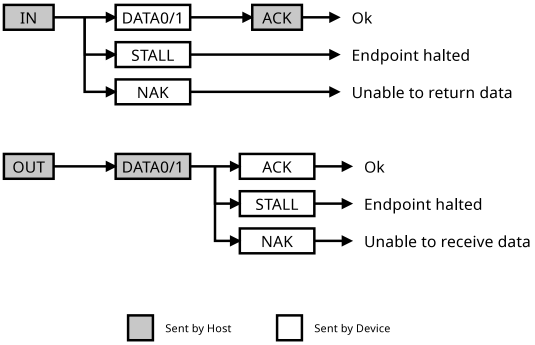
- Multiple devices using hubs
- Hot-plug devices
- Discoverable devices
- Each device has a unique address assigned by the host

Communication flow

- USB transfers are initiated by the Host.
- A USB transfer is made of bus transactions
- Most bus transactions involve the transmission of up to three packets
  - Token (IN, OUT, SETUP, ...)
    - First packet in a transaction
    - Identify transaction type and direction.
    - Identify transaction recipient (USB device address, Endpoint number)
  - Data (DATA0, DATA1, ...)
    - Contains the data related to the transaction
    - Can be an empty data packet (zero-length packet)
    - `wMaxPacketSize` size limit on each Endpoint (max 1024 bytes)
  - Handshake (ACK, NAK, STALL, ...)
    - Indicates whether the transfer was successful.

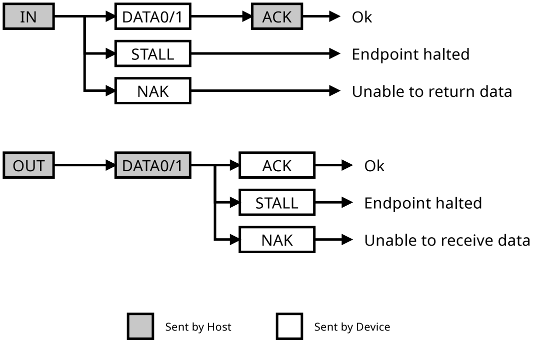
Bulk transfers

- Guaranteed delivery
- No guarantee of bandwidth or latency



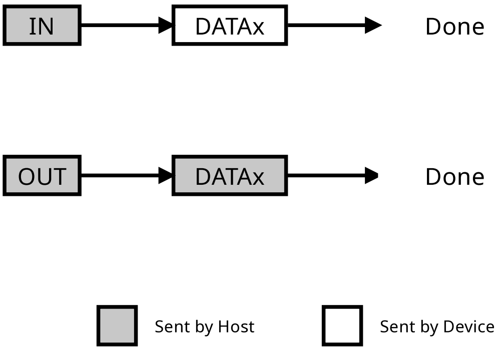
Interrupt transfers

- Guaranteed maximum service period
- Retry on next period in case of delivery failure
- Interrupt: periodic polling from the host



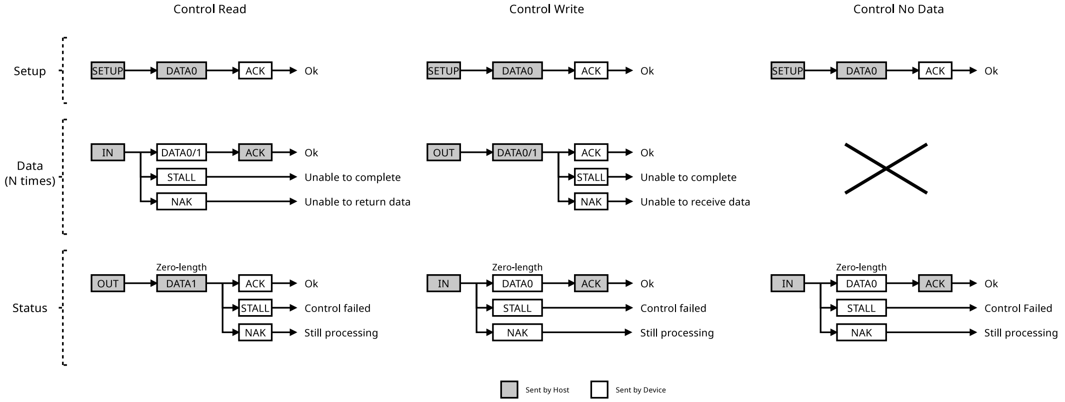
Isochronous transfers

- Guaranteed bandwidth and data rate
- No retry in case of delivery failure



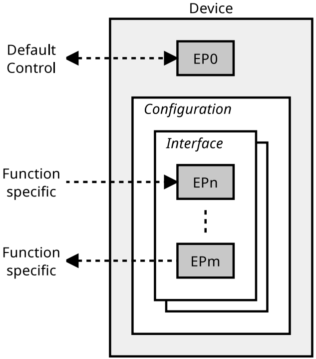
Control transfers

- Intended to support configuration/command/status communication flow



Endpoints, Interfaces, Configuration

- **Endpoint**: Terminus of a communication flow between the host and the device.
  - Uniquely referenced (device address, endpoint number, direction)
  - EP0 (IN/OUT): Used to configure and control the device (mandatory).
  - Endpoints other than EP0 are function specific.
- **Interface**: Group of endpoints to provide a function.
  - Several interfaces can be available at the same time (multi-function printer/scanner)
- **Configuration**: Device capabilities.
  - Power budget, remote wake-up support, number of interfaces.
  - One or more interfaces are present in each configuration.
  - Only one configuration can be activated.



Standard requests

- Use control transfers through EP0.

Table 9-2. Format of Setup Data

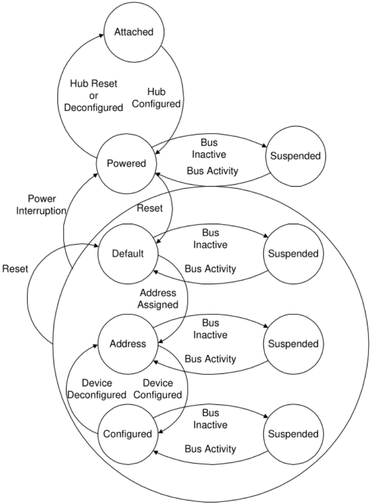
Offset	Field	Size	Value	Description
0	*bmRequestType*	1	Bitmap	Characteristics of request: D7: Data transfer direction 0 = Host-to-device 1 = Device-to-host D6...5: Type 0 = Standard 1 = Class 2 = Vendor 3 = Reserved D4...0: Recipient 0 = Device 1 = Interface 2 = Endpoint 3 = Other 4...31 = Reserved
1	*bRequest*	1	Value	Specific request (refer to Table 9-3)
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 9-3. Standard Device Requests

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00000000B 00000001B 00000010B	CLEAR_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
10000000B	GET_CONFIGURATION	Zero	Zero	One	Configuration Value
10000000B	GET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
10000001B	GET_INTERFACE	Zero	Interface	One	Alternate Interface
10000000B 10000001B 10000010B	GET_STATUS	Zero	Zero Interface Endpoint	Two	Device, Interface, or Endpoint Status
00000000B	SET_ADDRESS	Device Address	Zero	Zero	None
00000000B	SET_CONFIGURATION	Configuration Value	Zero	Zero	None
00000000B	SET_DESCRIPTOR	Descriptor Type and Descriptor Index	Zero or Language ID	Descriptor Length	Descriptor
00000000B 00000001B 00000010B	SET_FEATURE	Feature Selector	Zero Interface Endpoint	Zero	None
00000001B	SET_INTERFACE	Alternate Setting	Interface	Zero	None
10000010B	SYNCH_FRAME	Zero	Endpoint	Two	Frame Number

Extracted from the USB2.0 standard

Device states



Extracted from the USB2.0 standard

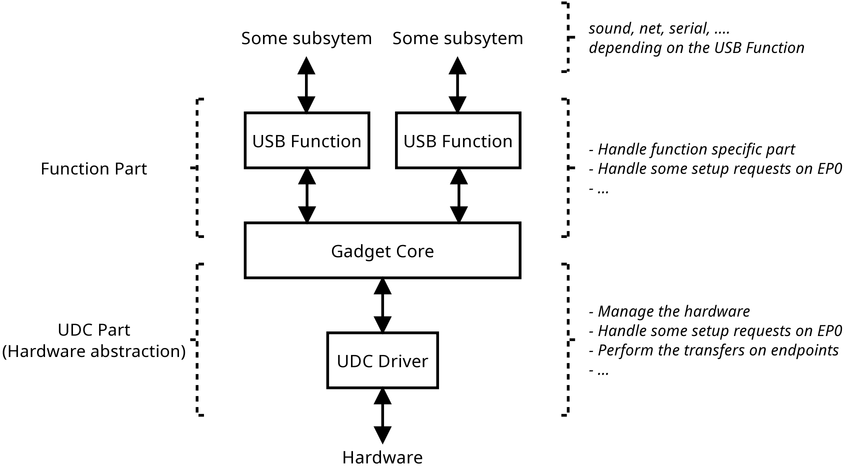
State	*Description*
Attached	Device is attached to the USB but is not powered.
Powered	The host set VBUS. Device is powered but has not received the USB reset. It must not answer to any transaction.
Default	Device received the USB reset. It is addressable at the default USB address (address 0) and answers to transaction on EP0 using this USB address.
Address	The host assigned a unique device USB address using a SET_ADDRESS request. Device is addressable at the address assigned and answers to transactions on EP0 using this unique USB address.
Configured	Device received a SET_CONFIGURATION request. Device is ready to use, its functions are available and it answers to the transactions on all EPs.
Suspended	Device enter the suspend state when it has observed no bus activities for a specified period. In suspended state, it maintains its internal status including its address and configuration. Suspended state is exited when there is bus activity.

Bus enumeration

- Detect devices, assign addresses, configure.
- Communication through standard requests.
- Enumeration sequence (somewhat simplified)
  1. Device plugged to a powered port (the port is disabled).
     - VBUS is available at the device side.
  2. The device connects the pull-up data line resistor.
     - The Hub detects the attachment and informs the host (status change).
  3. The host asks the Hub for a port enable and a reset on that port.
     - Following the reset, the device is in the Default state.
     - It answers to the default address and EP0 is accessible.
  4. The host assigns a unique address (SET_ADDRESS request).
     - The device is in the Address state.
     - It answers to the assigned address and EP0 is accessible.
  5. The host read the descriptors (GET_DESCRIPTOR requests)
  6. The host configure the device (SET_CONFIGURATION request).
     - The device is in the Configured state.
     - The interfaces available in the selected configuration and their endpoints are accessible.

Linux USB Device Controller (UDC)

USB gadget (Linux as an USB device)



UDC driver structure

- Includes `<linux/usb/gadget.h>`
  - The Chapter 9 USB2.0 standard references (`<uapi/linux/usb/ch9.h>`) included.
- Provides hooks for device management (`struct usb_gadget_ops`)
- Provides hooks for endpoints management (`struct usb_ep_ops`)
- Uses functions from the Gadget Core API for
  - Registering the UDC,
  - Signaling USB events,
  - Forwarding EP0 requests to the Gadget core,
  - Signaling Endpoints end of transfers.

Gadget ops

linux/usb/gadget.h (simplified, only basic hooks extracted)

```
struct usb_gadget_ops {
        // ...
        int (*pullup) (struct usb_gadget *, int is_on);
        // ...
        int (*udc_start)(struct usb_gadget *,
                         struct usb_gadget_driver *);
        int (*udc_stop)(struct usb_gadget *);
        // ...
};
```

Gadget ops - udc_start() / udc_stop()

- `udc_start()`: Start the UDC
  - The UDC driver is going to be used and needs to start.
  - Start VBUS monitoring (if possible)
  - No USB transfer should be enabled at this time
- `udc_stop()`: Stop the UDC
  - The UDC driver is not used anymore.
  - No more events can be signaled by the UDC.

Gadget ops - pullup()

- `pullup()`: Activate or deactivate the data line pull-up
  - The `pullup()` is called by the Gadget core after the VBUS detection is signaled.
  - Activate (is_on != 0)
    - Connect the pull-up.
    - USB connect detected by the host → Beginning of USB activities (Bus enumeration).
  - Deactivate (is_on == 0)
    - Disconnect the pull-up
    - USB disconnect detected by the host → End of USB activities.
    - No transfer anymore.

Endpoint ops

linux/usb/gadget.h (simplified, only basic hooks extracted)

```
struct usb_ep_ops {
        int (*enable) (struct usb_ep *ep,
                       const struct usb_endpoint_descriptor *desc);
        int (*disable) (struct usb_ep *ep);
        // ...
        struct usb_request *(*alloc_request) (struct usb_ep *ep,
                                              gfp_t gfp_flags);
        void (*free_request) (struct usb_ep *ep, struct usb_request *req);

        int (*queue) (struct usb_ep *ep, struct usb_request *req,
                      gfp_t gfp_flags);
        int (*dequeue) (struct usb_ep *ep, struct usb_request *req);

        int (*set_halt) (struct usb_ep *ep, int value);
        int (*set_wedge) (struct usb_ep *ep);
        // ...
};
```

Endpoint ops - enable() / disable()

Endpoint chosen by the core among the available Endpoint list.

- `enable()`: Enable the endpoint
  - Setup the endpoint based on `struct usb_endpoint_descriptor`
  - Configure the hardware to handle the endpoint.
- `disable()`: Disable the endpoint
  - Disable the endpoint at the hardware level.
  - Complete all pending requests (`usb_gadget_giveback_request()` with `req.status = -ESHUTDOWN`)
  - The endpoint will not be used anymore.

EP0 is always enabled, never disabled.

Endpoint ops - set_halt() / set_wedge()

- `set_halt()`: Set or clear the endpoint halt feature.
  - Halted endpoint will return a STALL
  - The Host `GET_STATUS(endpoint)` request returns the halt status.
  - The Host `CLEAR_FEATURE(HALT_ENDPOINT)` request clears the halt state.
- `set_wedge()`: Set the endpoint halt feature.
  - Same as `set_halt(ep, 1)` except:
  - The Host `CLEAR_FEATURE(HALT_ENDPOINT)` request **does not** switch the endpoint to its normal state.
  - Only a `set_halt(ep, 0)` can clear the halt state.

Request

- Data exchanged using an endpoint (`struct usb_request`)
- Chained using a queue per endpoint
- IN endpoint (from device to host): Data to send.
  - One request $\rightarrow$ One or more data packet (max packet size).
  - Zero length packet can be added if needed.
- OUT endpoint (from host to device): Data received.
  - Merge received data packets up to the request size
  - Zero length packet or short packet terminates the request
  - Data Packet received cannot be split over several requests.
- Give back to the Core using `usb_gadget_giveback_request()`.

Endpoint ops - alloc_request() / free_request()

- `alloc_request()`: Allocate a request
  - One or more requests can be allocated per endpoint.
  - Can setup extra resources (DMA buffer)
  - An allocated request can be used several times (i.e. queued and completed several times)
- `free_request()`: Free a request
  - The request is no longer used
  - Release specific hardware request resources.
  - Free the request

Endpoint ops - queue() / dequeue()

- `queue()`: Queue a request
  - The request is queued to be processed.
  - Automatically removed from queue at the end of processing.
  - `usb_gadget_giveback_request()` called at the end of processing.
  - Start the queue processing if not already done.
- `dequeue()`: Dequeue a request
  - Dequeue an queued request.
  - Complete the request (`usb_gadget_giveback_request()` with `req.status = -ECONNRESET`).
  - Was the first in queue? → Start processing the next request.

Core API

`linux/usb/gadget.h` (simplified and commented, only basic functions extracted)

```
/* Register the UDC */
extern int usb_add_gadget_udc(struct device *parent, struct usb_gadget *gadget);

/* Unregister the UDC */
extern void usb_del_gadget_udc(struct usb_gadget *gadget);

/* Notify the VBUS status, and try to connect or disconnect gadget */
extern void usb_udc_vbus_handler(struct usb_gadget *gadget, bool status);

/* Notify the Core that a bus reset occurs */
extern void usb_gadget_udc_reset(struct usb_gadget *gadget,
                                 struct usb_gadget_driver *driver);

/* Set gadget state */
extern void usb_gadget_set_state(struct usb_gadget *gadget, enum usb_device_state state);

/* Give a request back to the Core layer */
extern void usb_gadget_giveback_request(struct usb_ep *ep, struct usb_request *req);

struct usb_gadget_driver {
        //...
        /* Handle EP0 control requests */
        int     (*setup)(struct usb_gadget *, const struct usb_ctrlrequest *);
        //...
};
```

myudc driver data - data & ops

```
struct myudc_ep {
        struct usb_ep ep;
        struct list_head queue;
        struct myudc *myudc;
        u8 id;
        bool disabled;
        //...
};

struct myudc {
        struct usb_gadget gadget;
        struct usb_gadget_driver *driver;
        /* My udc hardware supports 8 Endpoints */
        struct myudc_ep ep[8];
        //...
};

//...
```

```
static static struct usb_ep_ops myudc_ep_ops = {
        .enable = myudc_ep_enable,
        .disable = myudc_ep_disable,
        .queue = myudc_ep_queue,
        .dequeue = myudc_ep_dequeue,
        .set_halt = myudc_ep_set_halt,
        .set_wedge = myudc_ep_set_wedge,
        .alloc_request = myudc_ep_alloc_request,
        .free_request = myudc_ep_free_request,
}

...

static struct usb_gadget_ops myudc_gadget_ops = {
        .udc_start = myudc_udc_start,
        .udc_stop = myudc_udc_stop,
        .pullup = myudc_pullup,
};
```

myudc driver data - endpoints information

```
struct myudc_ep_info {
        const char *name;
        struct usb_ep_caps caps;
        u16 maxpacket_limit;
};

#define EP_INFO(_name, _caps, _maxpacket_limit)      \
        {                                            \
                .name = _name,                       \
                .caps = _caps,                       \
                .maxpacket_limit = _maxpacket_limit, \
        }

/* Available endpoints (from hardware datasheet) */
static const struct myudc_ep_info myudc_ep_info[8] = {
        [0] = EP_INFO("ep0-ctrl",
                      USB_EP_CAPS(USB_EP_CAPS_TYPE_CONTROL, USB_EP_CAPS_DIR_ALL),
                      64),
        [1] = EP_INFO("ep1-bulk",
                      USB_EP_CAPS(USB_EP_CAPS_TYPE_BULK, USB_EP_CAPS_DIR_ALL),
                      512),
        // ...
        [4] = EP_INFO("ep5-int",
                      USB_EP_CAPS(USB_EP_CAPS_TYPE_INT, USB_EP_CAPS_DIR_ALL),
                      1024),
        // ...
        [7] = EP_INFO("ep7-iso",
                      USB_EP_CAPS(USB_EP_CAPS_TYPE_ISO, USB_EP_CAPS_DIR_ALL),
                      1024),
};
```

- Used during `probe()` call to initialize endpoints.
- Endpoint name
  - Format `"epN*"` with `N` the endpoint number
- Endpoint capabilities `USB_EP_CAPS()`
- Endpoint max packet size limit

UDC driver probe()

- Initialize gadget fields (name, max_speed, ops)
- Initialize available endpoints
  - Disabled (will be enabled later)
  - Initialize the request queue
  - Initialize endpoint fields (name, ops)
  - Set the endpoint capabilities (caps)
  - Set the maximum packet size limit (usb_ep_set_maxpacket_limit())
- Set the specific EP0
- Set the available endpoint list (endpoints other than EP0)
- Register the UDC driver

```c
static int myudc_probe(struct platform_device *pdev)
{
        struct myudc_ep *myudc_ep;
        struct myudc *myudc;

        // 1. Allocate myudc
        myudc = devm_kzalloc(dev, sizeof(*myudc), GFP_KERNEL);
        // ...
        // 2. Initialize gadget fields
        myudc->gadget.name = "myudc";
        myudc->gadget.max_speed = USB_SPEED_HIGH;
        myudc->gadget.ops = &myudc_gadget_ops;
        // 3. Initialize endpoints
        INIT_LIST_HEAD(&myudc->gadget.ep_list);
        for (i = 0; i < ARRAY_SIZE(myudc->ep); i++) {
                myudc_ep = &myudc->myudc_ep[i];
                INIT_LIST_HEAD(&myudc_ep->queue);
                myudc_ep->id = i;
                myudc_ep->disabled = 1;
                myudc_ep->myudc = myudc;
                myudc_ep->ep.ops = &myudc_ep_ops;
                myudc_ep->ep.name = myudc_ep_info[i].name;
                myudc_ep->ep.caps = myudc_ep_info[i].caps;
                usb_ep_set_maxpacket_limit(&myudc_ep->ep,
                                           myudc_ep_info[i].maxpacket_limit);
                // ...
                if (myudc_ep->id == 0) {
                        //4.a Set the specific EP0
                        myudc->gadget.ep0 = &myudc_ep->ep;
                } else {
                        //4.b Add the endpoint to the available endpoint list
                        INIT_LIST_HEAD(&myudc_ep->ep.ep_list);
                        list_add_tail(&myudc_ep->ep.ep_list,
                                      &myudc->gadget.ep_list);
                }
        }
        // ...
        // 5. Register the UDC driver
        return usb_add_gadget_udc(dev, &myudc->gadget);
}
```

Startup & Reset - VBUS en USB reset events

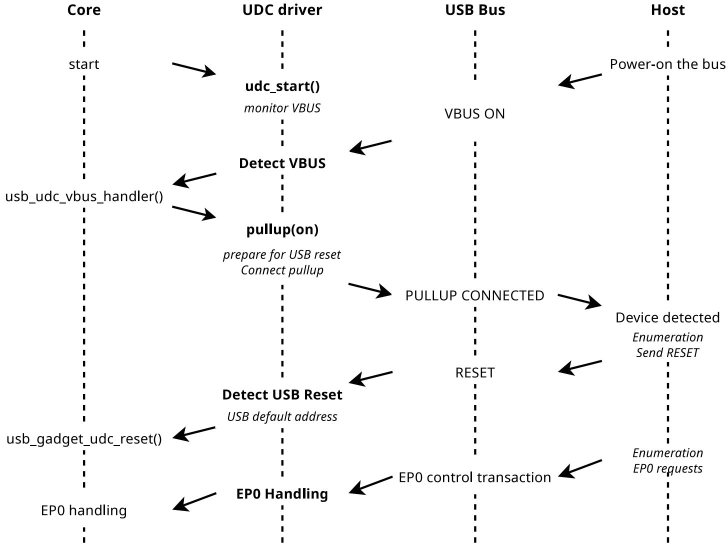
- VBUS change events
  - Signal event to the core.
    - it will call pullup (on/off).

```
static void myudc_handler_vbus(struct myudc *myudc)
{
	bool is_vbus;
	// ...
	is_vbus = my_udc_get_vbus(myudc);
	if (is_vbus) {
		usb_udc_vbus_handler(&myudc->gadget, true);
		usb_gadget_set_state(&myudc->gadget, USB_STATE_POWERED);
	} else {
		usb_udc_vbus_handler(&myudc->gadget, false);
		usb_gadget_set_state(&myudc->gadget, USB_STATE_NOTATTACHED);
	}
	//...
}
```

- USB reset events
  - Complete all pending requests
  - Speed negotiated during USB reset
  - Reset the address to the USB default address.
  - Signal the reset to the core
  - Only EP0 is available after a reset

```
static void myudc_handler_usb_reset(struct myudc *myudc)
{
	// ...

	for (i = 0; i < ARRAY_SIZE(myudc->ep); i++)
		myudc_ep_nuke(&myudc->ep[i], -ESHUTDOWN);

	myudc_disable_all_endpoints(myudc);

	/* Set speed */
	udc->gadget.speed = myudc_is_high_speed(myudc) ?
		USB_SPEED_HIGH : USB_SPEED_FULL;

	/* Use USB default address */
	myudc_set_usb_address(myudc, 0x00);

	/* Setup endpoint zero */
	myudc_ep0_setup(myudc);

	/* Signal the reset to the core */
	if (myudc->driver)
		usb_gadget_udc_reset(&myudc->gadget, myudc->driver);
}
```

Startup & Reset - Putting all together



EP0 Control requests handling

- Fully handled at UDC Level for some control requests
  - `SET_ADDRESS`: Set the unique USB address
  - `GET_STATUS(Device)`: Remote WakeUp, Self powered
  - `GET_STATUS(Endpoint)`: Endpoint Halt state
  - `{SET,CLEAR}_FEATURE(Device)`: Remote WakeUp
  - `{SET,CLEAR}_FEATURE(Endpoint)`: Endpoint Halt state
- Delegate to the Core for others
  `ret = myudc->driver->setup(&myudc->gadget, ctrlrequest)`
  - The Core performs the related operations
  - Queue a request in the EP0 queue for the data or status stage
    - data stage if data (IN or OUT) are needed (control read/write)
    - status stage if no data are needed (control without data)
  - Returns `USB_GADGET_DELAYED_STATUS` if status stage request will be queued later.
- Process queue (IN or OUT) if needed.
- Don't forget the status stage (send/receive Zero length packet)

Other EP handling

- Data processing done at the Core/Function level
- The UDC performs the data transfers
  - Just process the EP queue according to the EP direction.

How to test?

testusb

- http://www.linux-usb.org/usbtest/
- A Bootlin blog post related to testusb (https://bootlin.com/blog/test-a-linux-kernel-usb-device-controller-driver-with-testusb/)
- Quite old tool
- Host part (test tooling)
  - Dedicated kernel driver: `usbtest.ko` (CONFIG_USB_TEST=m).
  - A user-space program that asks for test: `testusb` (kernel sources tools/usb/)
  - `usbtest.ko` can hang on some failures (reboot needed).
    Use it on a dedicated tool board, not your workstation.
- Target part (system under test)
  - Precomposed `g_zero` gadget is sufficient (CONFIG_USB_ZERO=m)
  - The UDC to be tested

testusb

- On the target

```
# modprobe g_zero
```

- On the host
  - Transfers other than isochronous

```
# modprobe usbtest
# testusb -a -v512
[ 220.276460] usbtest 2-1:3.0: TEST 0: NOP
[ 220.292316] usbtest 2-1:3.0: TEST 1: write 1024 bytes 8 times
[ 220.324711] usbtest 2-1:3.0: TEST 2: read 1024 bytes 8 times
...
[ 223.250355] usbtest 2-1:3.0: TEST 29: Clear toggle between bulk writes 8 times
#
```

  - Isochronous transfers (supported by g_zero running on the target)

```
# modprobe usbtest alt=1
# testusb -a -v512
...
#
```

Interesting precomposed gadget

- g_mass_storage (CONFIG_USB_MASS_STORAGE)
  - Halts some endpoints.
  - Useful to test the halt feature.
- g_ether (CONFIG_USB_ETH)
  - Uses transfer sizes that are not a multiple of MaxPacketSize.
  - Useful to test transfers spanned on multiple packets.
  - The last packet can be less than MaxPacketSize.
- g_serial (CONFIG_USB_G_SERIAL)
  - In a basic configuration, each byte sent is echoed.
  - Test very short packets
  - Easy to isolate transfers

g_mass_storage

Errors appear if the endpoint halt feature is not well implemented.

- On the target
  - Create a file for the mass storage
  - Load the gadget

```
# dd if=/dev/zero of=/tmp/storage.part bs=1M count=8
# modprobe g_mass_storage file=/tmp/storage.part
```

- On the host
  - New USB removable disk detected.
  - Format the disk.
  - Transfer files.

g_ether

One UDC USB request $\leftrightarrow$ one Ethernet packet.
The completed UDC USB request size = The Ethernet packet size.

- On the target
  - Load the gadget
  - Retrieve files from host

```
# modprobe g_ether
# wget http://192.168.0.106:8080/test_file.bin
```

- On the host
  - Run a web server to serve various file sizes.
  - Trace the Ethernet transfers (Wireshark).

g_serial

Each character typed on the host is echoed.
When your hit 'enter', the whole buffer is echoed.

- On the target
  - Load the gadget
  - Do a loopback

```
# modprobe g_serial
# cat /dev/ttyGS0 > /dev/ttyGS0
```

- On the host
  - Open and play with the TTY (picocom)

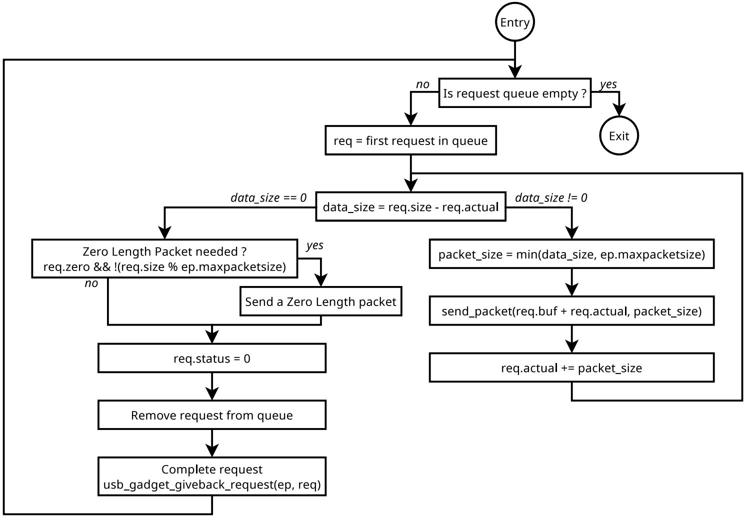
```
# picocom -b115200 /dev/ttyACM0
```

Questions? Suggestions? Comments?

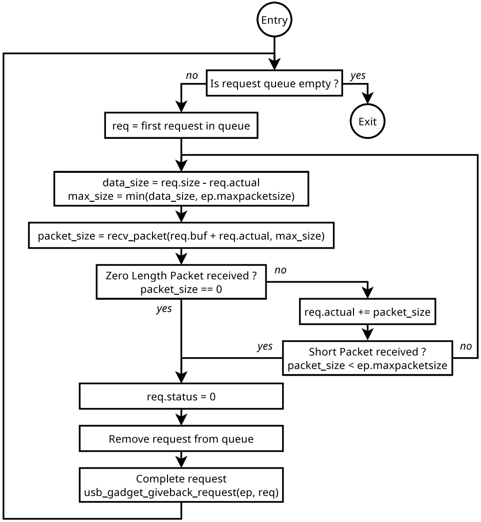
Hervé Codina
herve.codina@bootlin.com

Extra slides

Data transfers, IN queue processing (from device to host)



Data transfers, OUT queue processing (from host to device)



EP0 handling