Technology Consulting Company
Research, Development &
Global Standard

xHCI Driver Implementation

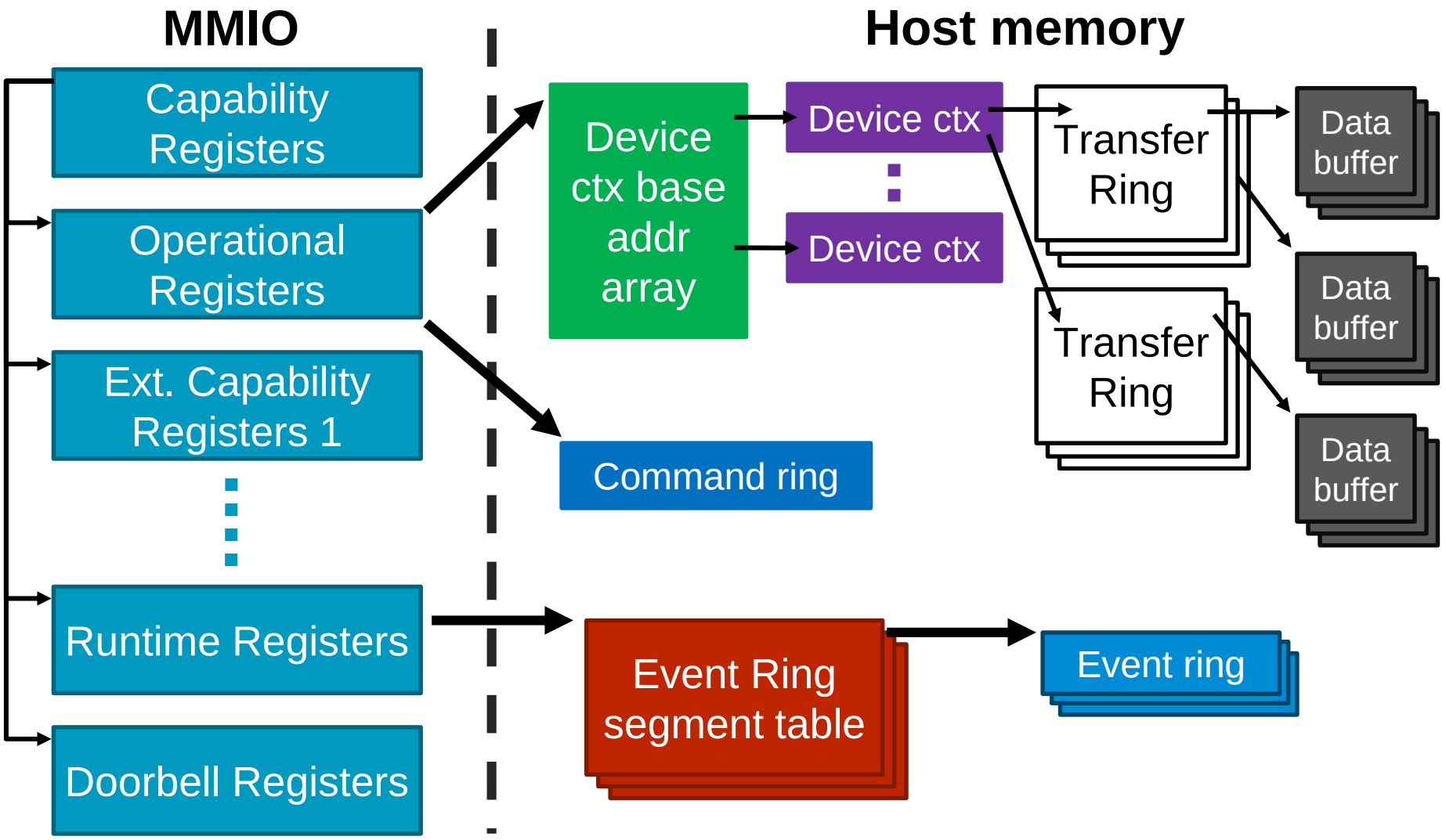
2016-11-30 @ BitVisor Summit 5

Ake Koomsin

Agenda

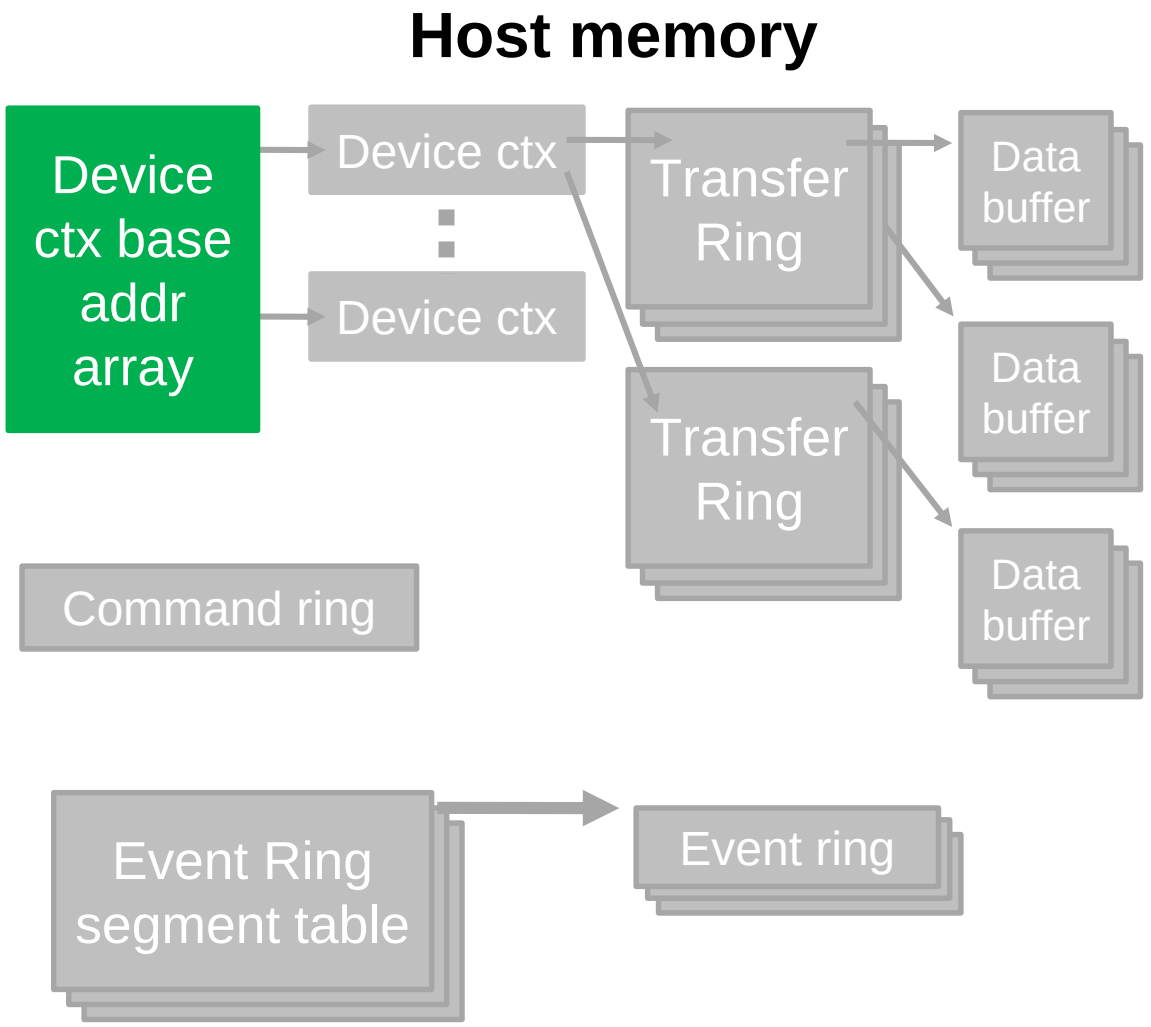
- xHCI Overview
- Relationship between BitVisor's host controller drivers and USB drivers
- Implementation Walkthrough
  - Changes in BitVisor

xHCI Overview

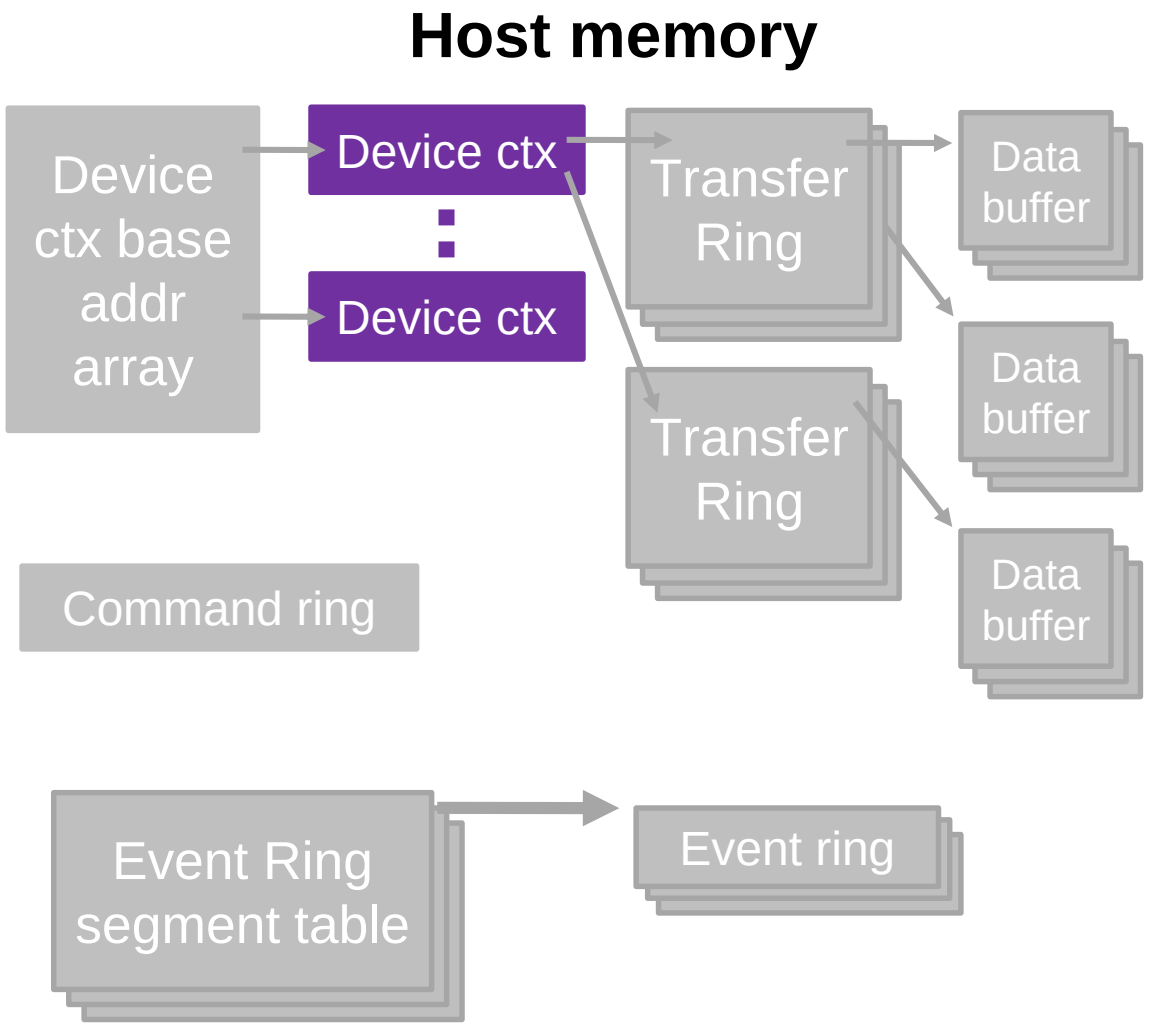


xHCI Overview

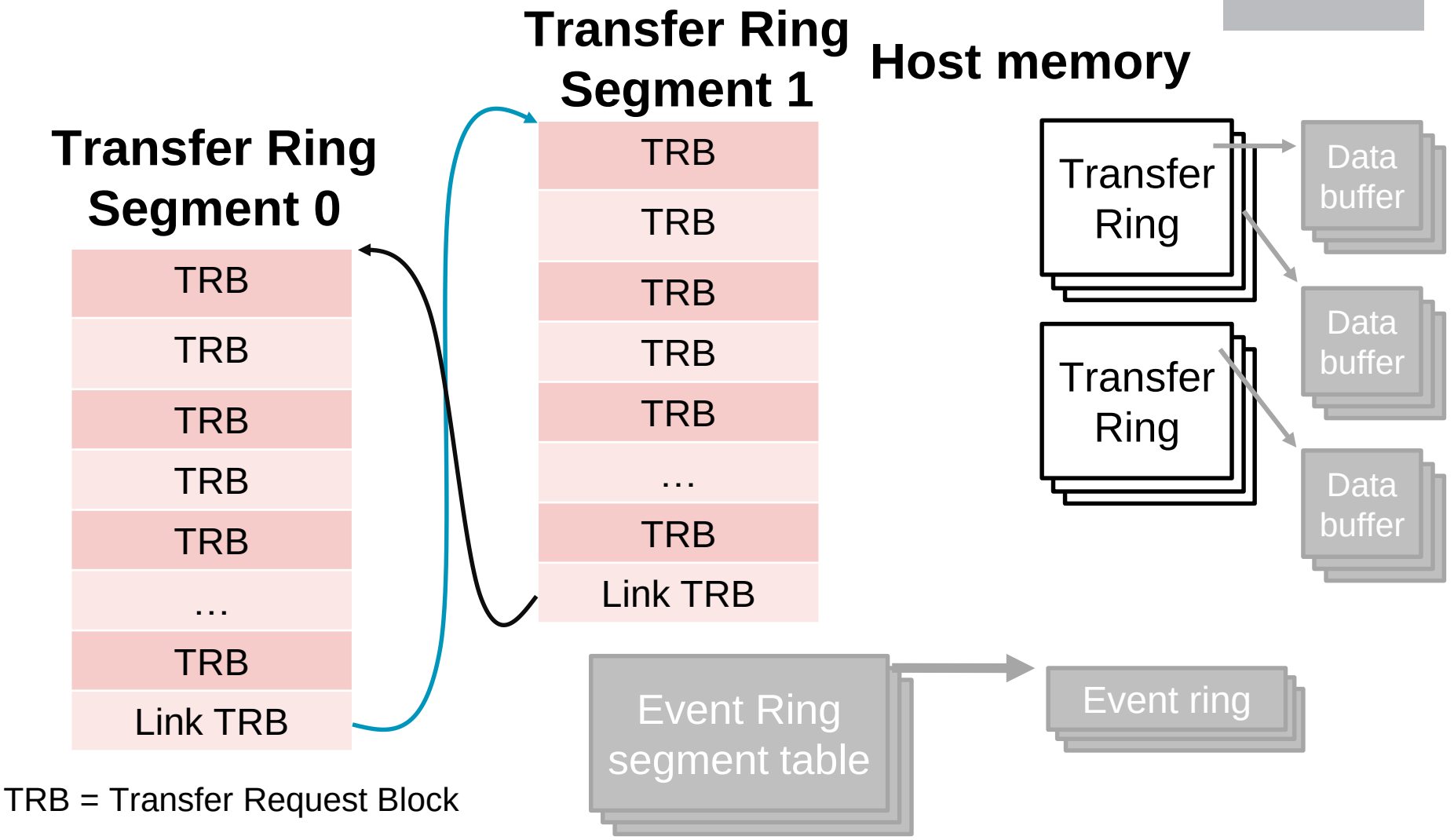
Index	Point to
0	Scratchpad
1	Dev Slot ID 1
2	Dev Slot ID 2
…	
N	Dev Slot ID N



xHCI Overview

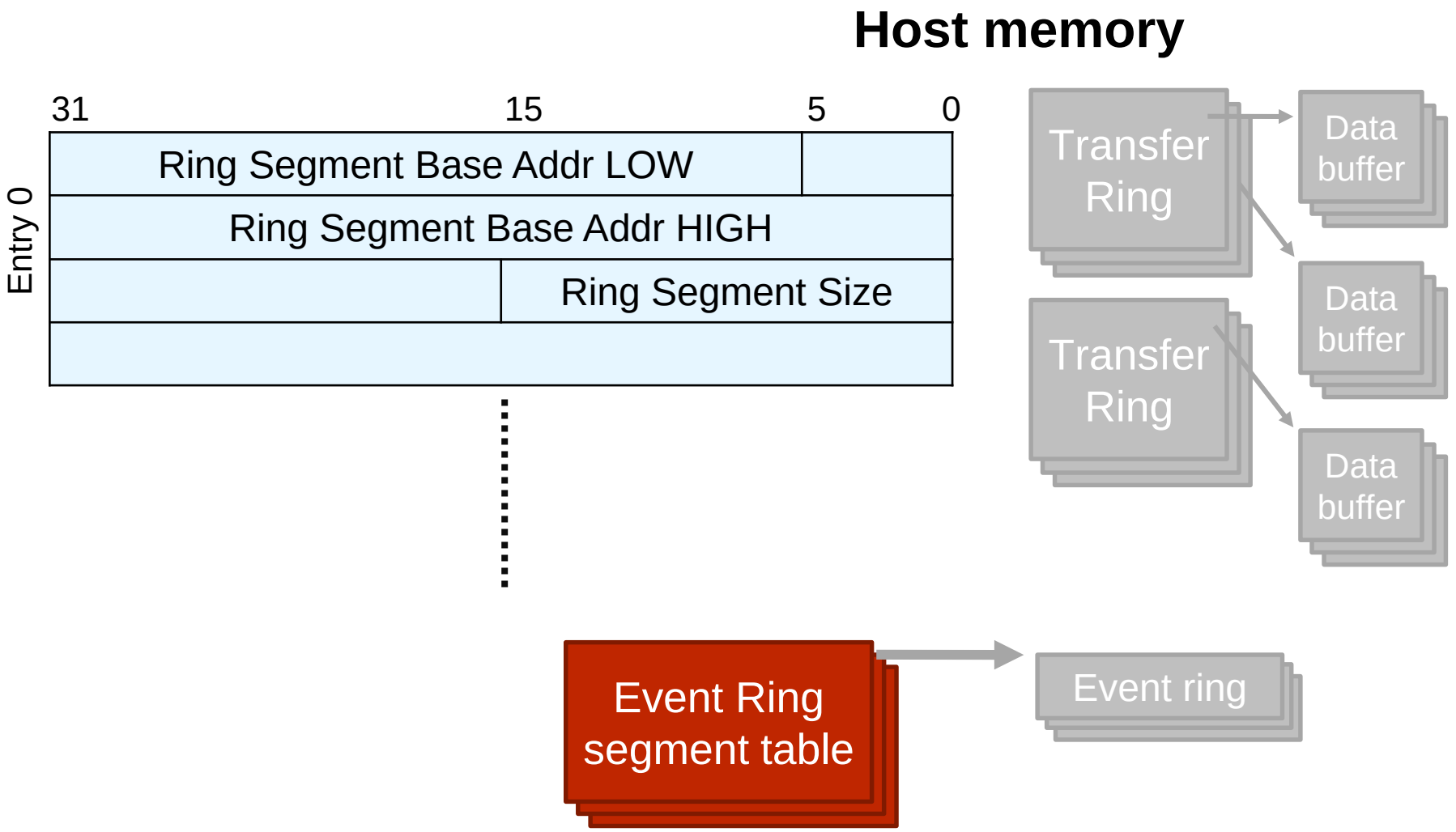


xHCI Overview

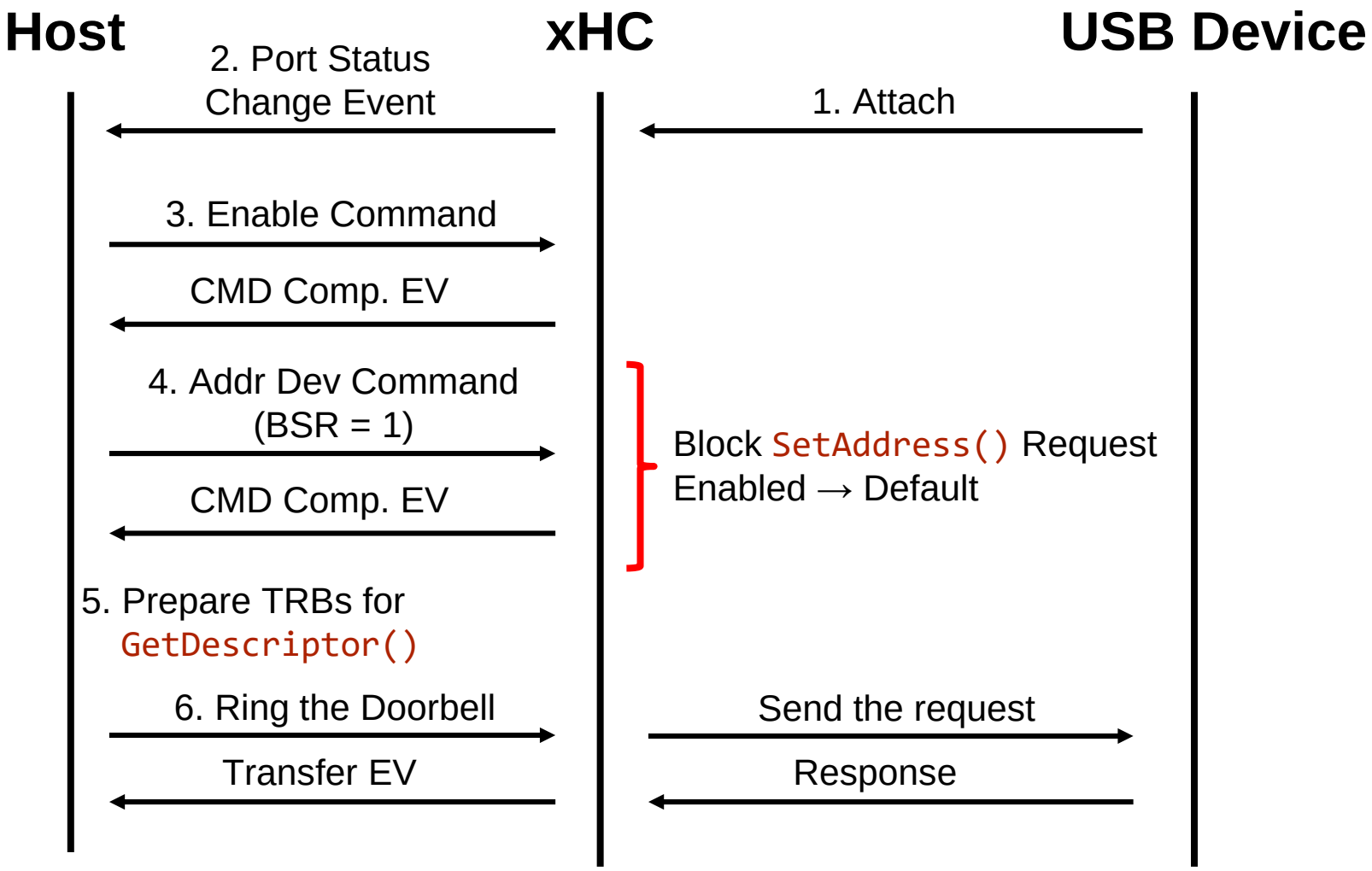


TRB = Transfer Request Block

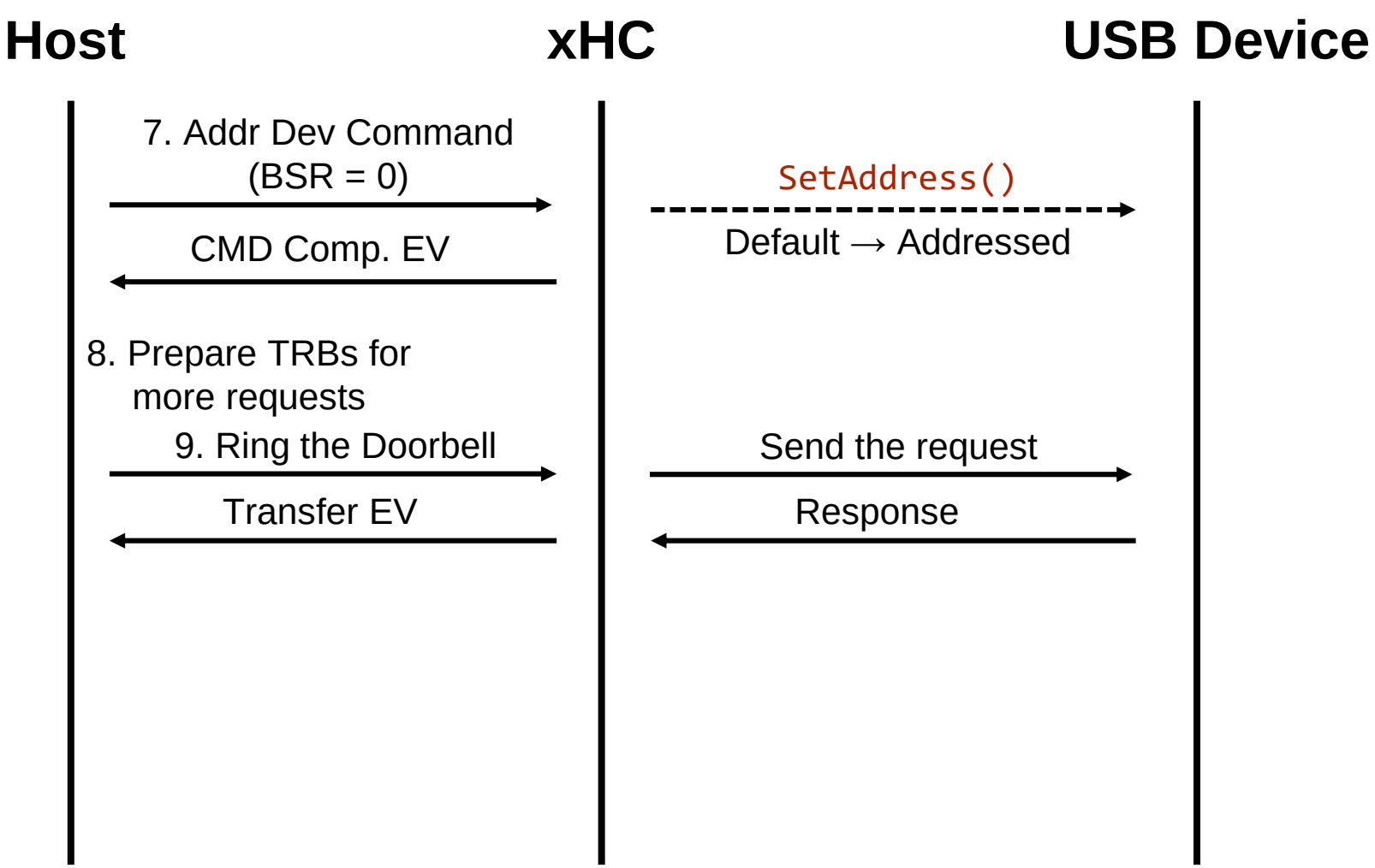
xHCI Overview



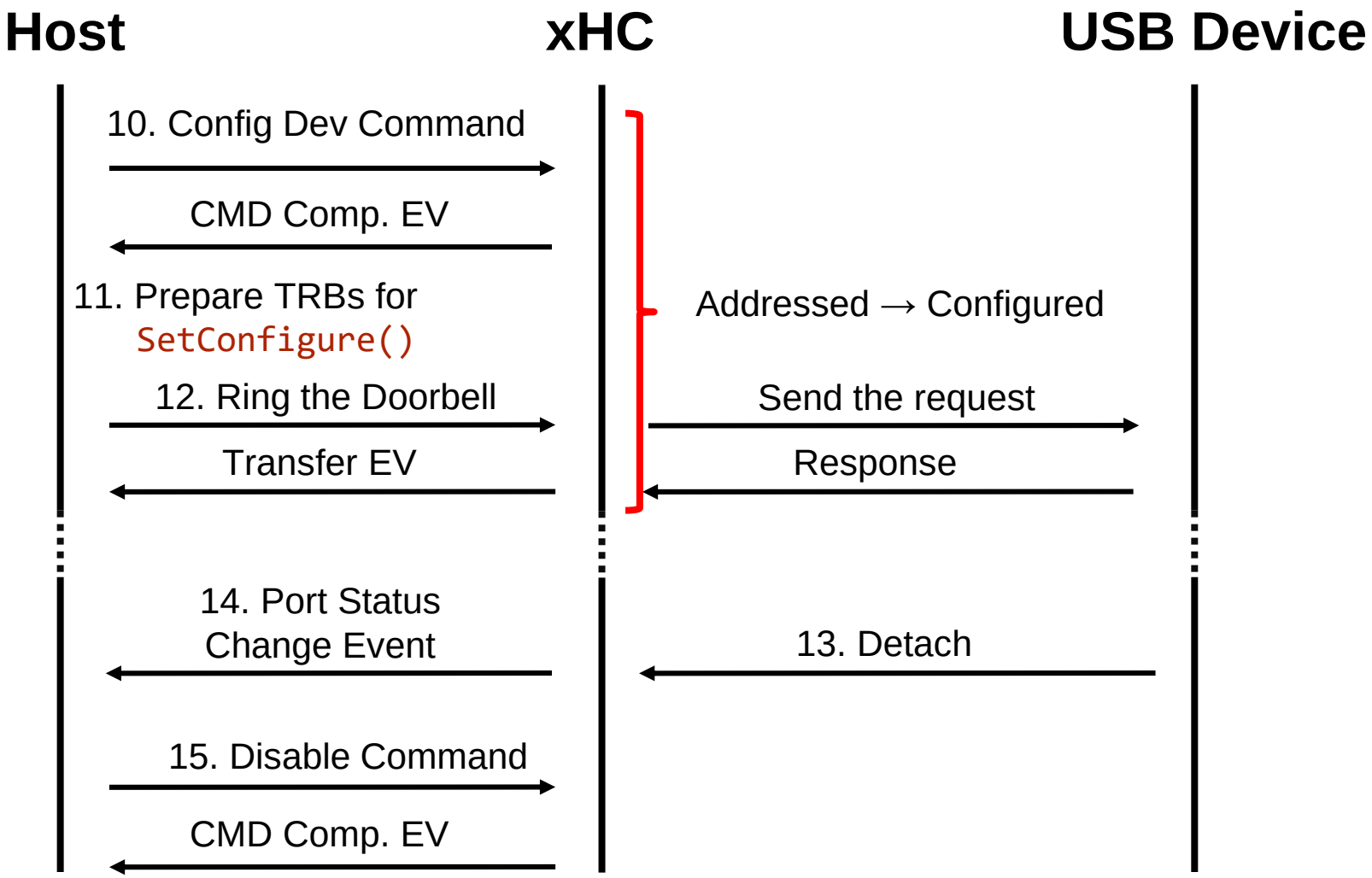
USB Device Initialization



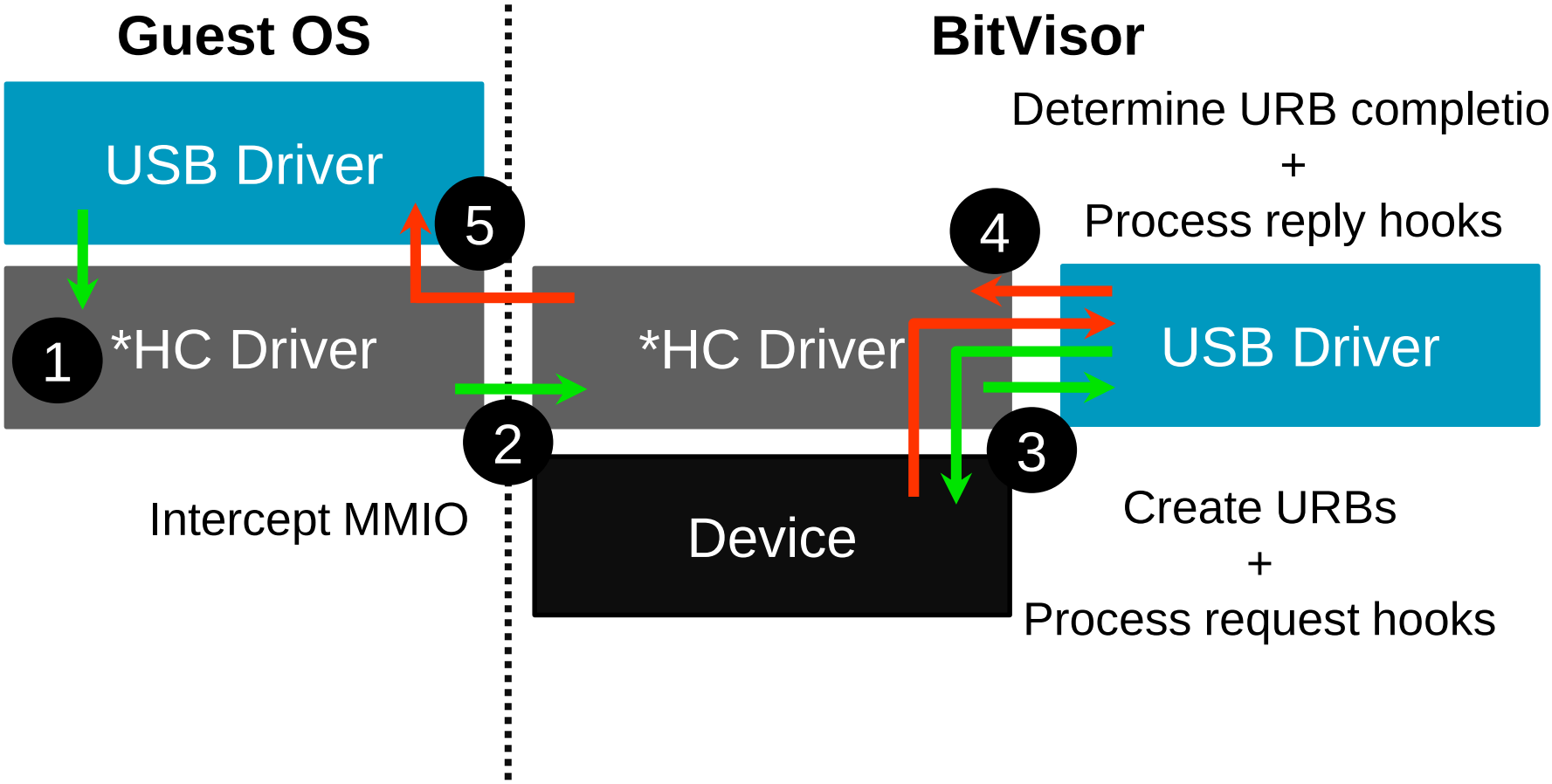
USB Device Initialization



USB Device Initialization



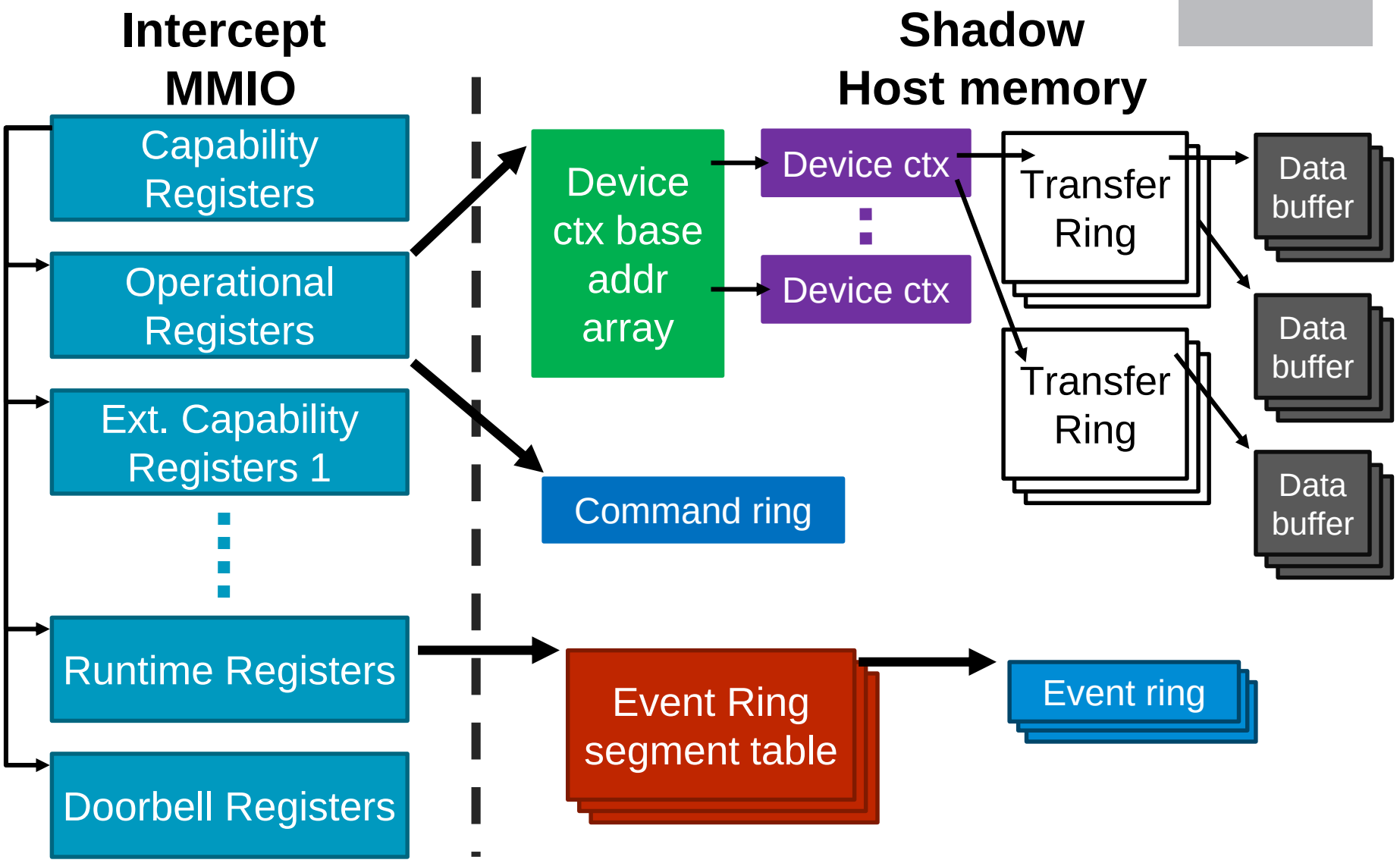
*HCI Driver & USB Driver



*HCI Driver & USB Driver

- *HCI Driver main responsibilities
  - Call `usb_register_host()` and initialize USB drivers
  - Construct USB Request Blocks (URBs) from data the guest OS submits
  - Pass URBs to USB Drivers by calling `usb_hook_process()`
  - Host Controller state synchronization
  - Provide a method for data shadowing
    - Not only data buffer, but also data structures used by the host controller

Implementation



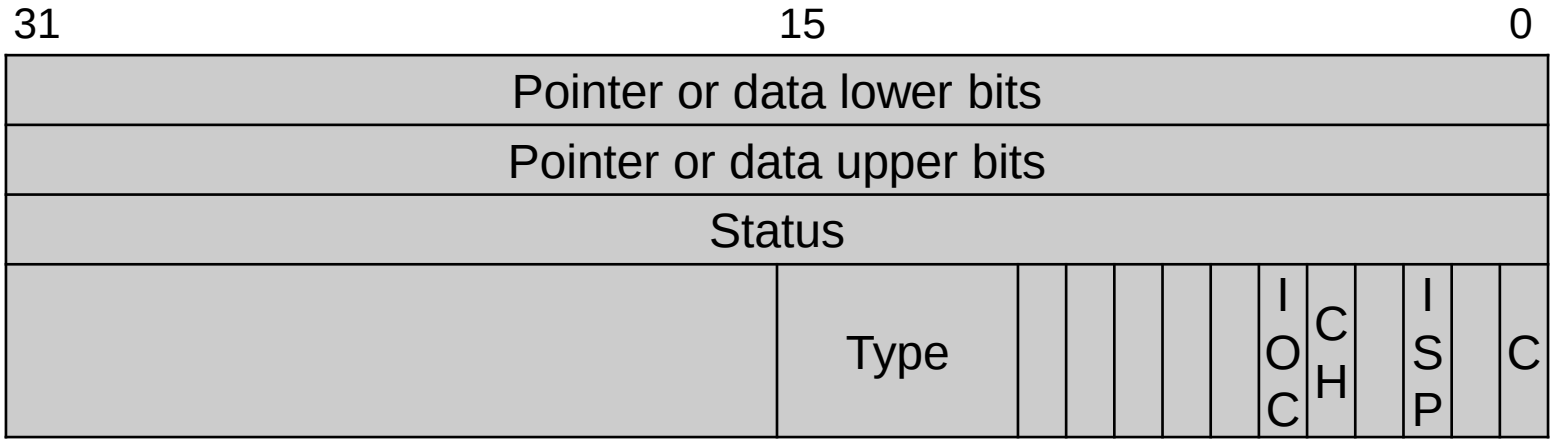
Implementation Outline

igel

- Intercept Operational Regs and Runtime Regs
  - Write : Replace the guest physical addresses with the host ones
  - Read : Return the guest physical addresses
- Intercept Doorbell Regs
  - Commands (Index 0)
  - USB Requests (Index 1 onwards)
- Synchronize states when there is an event

URB Construction

Data transfer related TRB

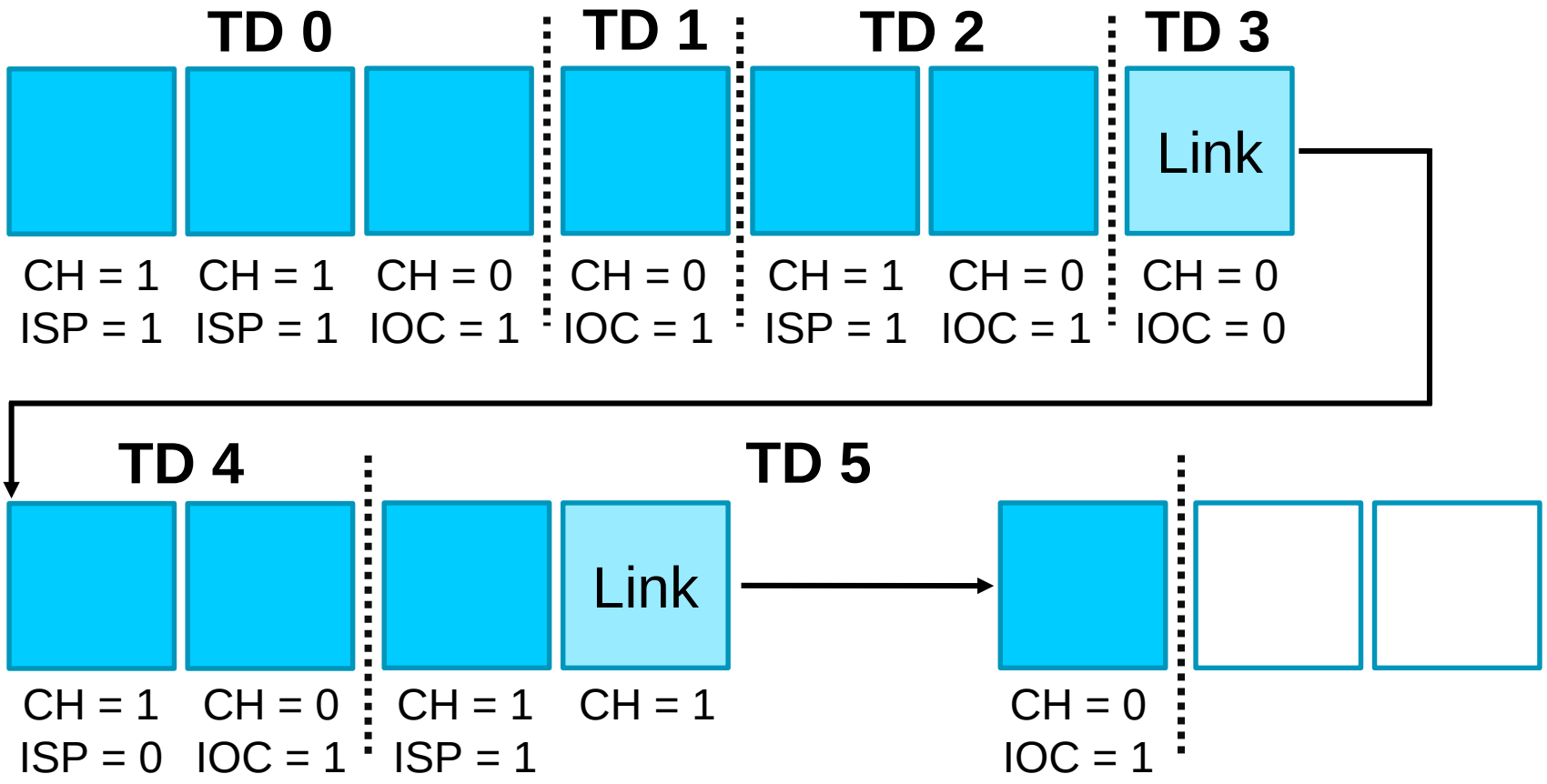


Type

1. Normal TRB
2. Setup Stage TRB
3. Data Stage TRB
4. Status Stage TRB
5. Isoch TRB
6. Link TRB
7. Event TRB

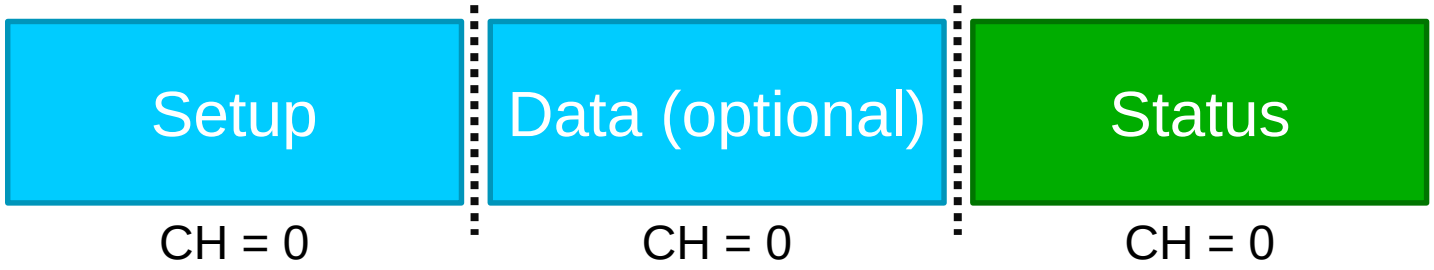
URB Construction

Transfer Descriptor (TD)

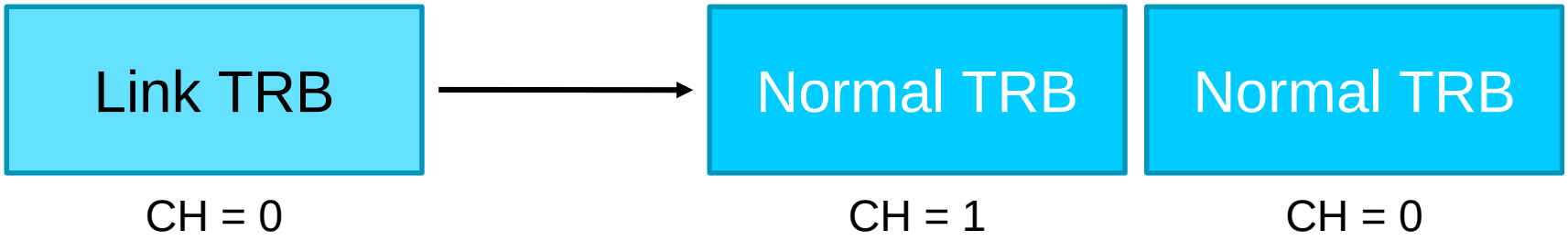


URB Construction

- One TD = One URB, with exceptions
  - USB Request TRBs, (3 TDs, 2 URBs in this example)



  - First TRB with not buffer



URB Construction


```
struct xhci_urb_private {
    u32 slot_id;
    u32 ep_no; /* Start from 0 */

    u32 start_idx;
    u32 end_idx;
    u32 next_td_idx;

    u32 start_seg;
    u32 end_seg;
    u32 next_td_seg;

    u8 start_toggle;
    u8 end_toggle;
    u8 next_td_toggle;

    u8 event_data_exist;

    u32 total_buf_size;

    struct usb_buffer_list *ub_tail;

    /* For URB completion */
    struct xhci_trb_meta *intr_tail;
    struct xhci_trb_meta *intr_list;

    /* For TR cloning */
    struct xhci_trb_meta *link_trb_tail;
    struct xhci_trb_meta *link_trb_list;

    struct xhci_trb *not_success_ev_trb;
};
```

URB Construction

```
static u8
handle_slot_write(...)
{
    ...
    struct usb_request_block *g_urb = construct_gurbs(host,
                                                      slot_id, ep_no);

    while (g_urb) {
        struct usb_request_block *h_urb = shadow_g_urb(g_urb);

        u8 ret = usb_hook_process(host->usb_host, h_urb,
                                  USB_HOOK_REQUEST);
        if (ret == USB_HOOK_DISCARD) {
            ...
        }

        append_h_urb_to_ep(h_urb, host, slot_id, ep_no);
        ...
        g_urb = g_urb->link_next;
    }
    ...
    return 1;
}
```

URB Construction

- We don’t know the number of TR segments and TR segment size at the beginning
  - We know only the base address of the first TR segment from Configure Device Command
  - Have to identify TR segment size and number of TR segments while traversing
  - Treat the Link TRB as the end of TR segments

URB Construction

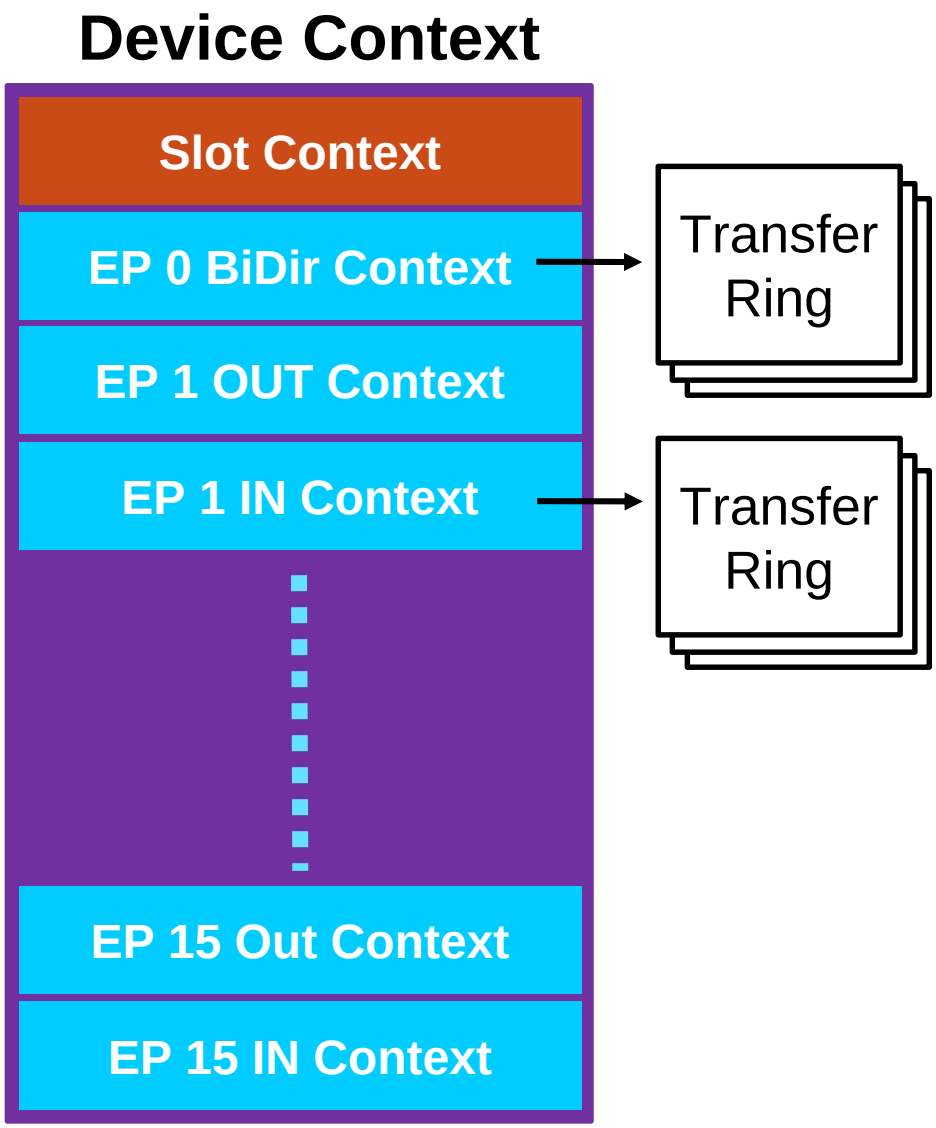
```
struct usb_request_block *
construct_gurbs(struct xhci_host *host, uint slot_id, uint ep_no)
{
    ...
    do {
        ...
        for (i_trb = current_idx; i_trb < n_trbs; i_trb++) {
            ...
            switch (type) {
                case XHCI_TRB_TYPE_LINK:
                    next_seg = get_next_seg(&g_trbs[i_trb], current_seg,
                                            h_ep_tr, g_ep_tr);
                    ... /* Preparing for traversing the next segment */
                    break;
                ...
            }
            ...
        }
    } while (!stop);
    ...
}
```

URB Construction

```
struct usb_request_block *
construct_gurbs(struct xhci_host *host, uint slot_id, uint ep_no)
{
    ...
    do {
        ...
        for (i_trb = current_idx; i_trb < n_trbs; i_trb++) {
            ...
            if (type == XHCI_TRB_TYPE_LINK) {
                /* Stop current scanning to start at the next segment */
                break;
            }

            if (i_trb + 1 == n_trbs && type != XHCI_TRB_TYPE_LINK) {
                g_ep_tr->tr_segs[current_seg].n_trbs *= 2;
                h_ep_tr->tr_segs[current_seg].n_trbs *= 2;
                g_ep_tr->current_idx = n_trbs;
                h_ep_tr->current_idx = n_trbs;
            }
        }
    } while (!stop);
    ...
}
```

EP Ownership



- Problems
  - BitVisor needs to obtain device info during devive initialization
  - Not all USB drivers need shadowing

EP Ownership

- BitVisor need to obtain device info during device initialization
  - It needs to own Endpoint 0 to do its job properly
  - For xHCI, there is no `SetAddress()` to intercept for USB address
    - xHC does this for us after receiving the Set Address Command (BSR = 0)
  - Current `usb_device_init_monitor()` implementation relies on intercepting `SetAddress()` response
    - This assumption is not valid anymore
  - The last interrupter in Runtime Registers is reserved for BitVisor
    - Polling the Event Rings for events

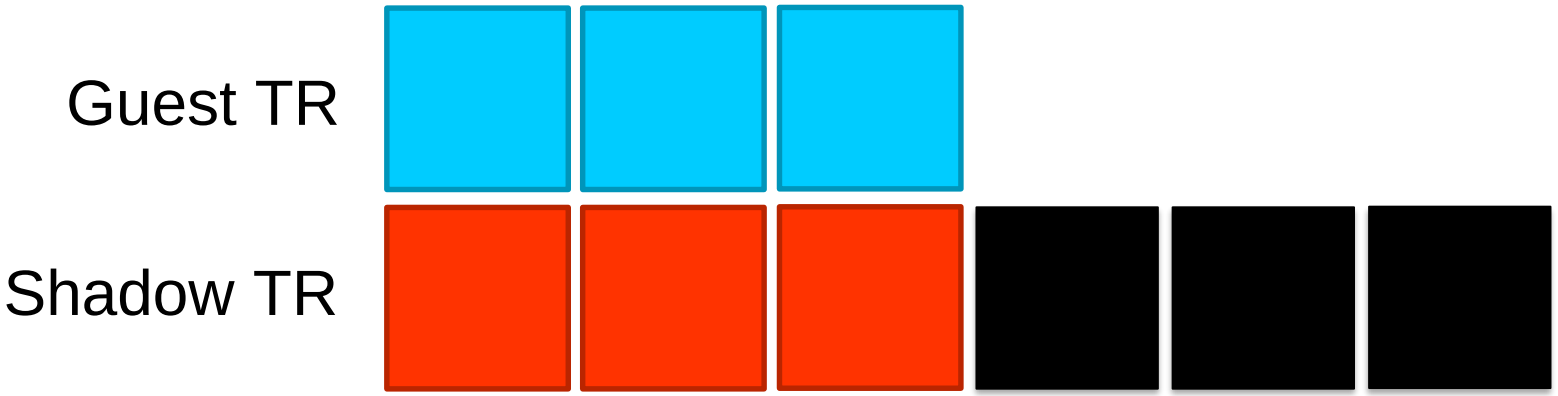
Changes in BitVisor (1)

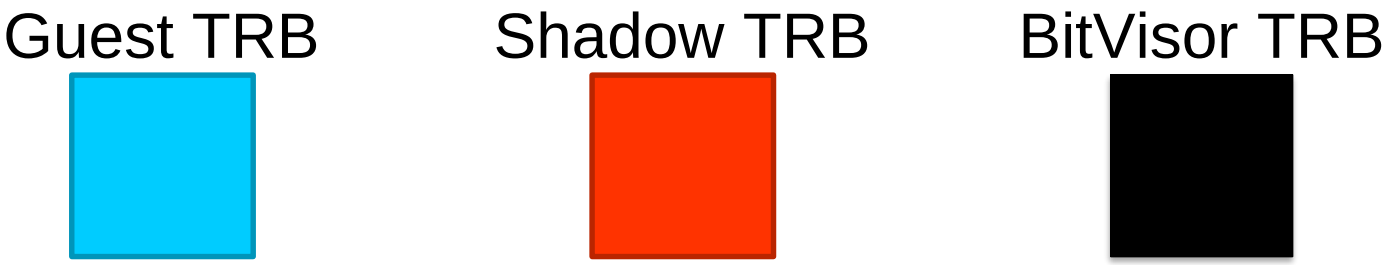
igel

- Changes in `usb_device_init_monitor()`
  - *HCI drivers are responsible for creating USB hooks for device monitoring from now
    - xHC does this for us after receiving the Address Device Command (BSR = 0)
- Changes in `struct usb_hook`
  - Introduce `before_callback`, `after_callback` to `struct usb_hook`
  - Introduce `exec_once` flag to `struct usb_hook`
    - If `exec_once` is set, the hook is removed after its execution

EP0 Ownership

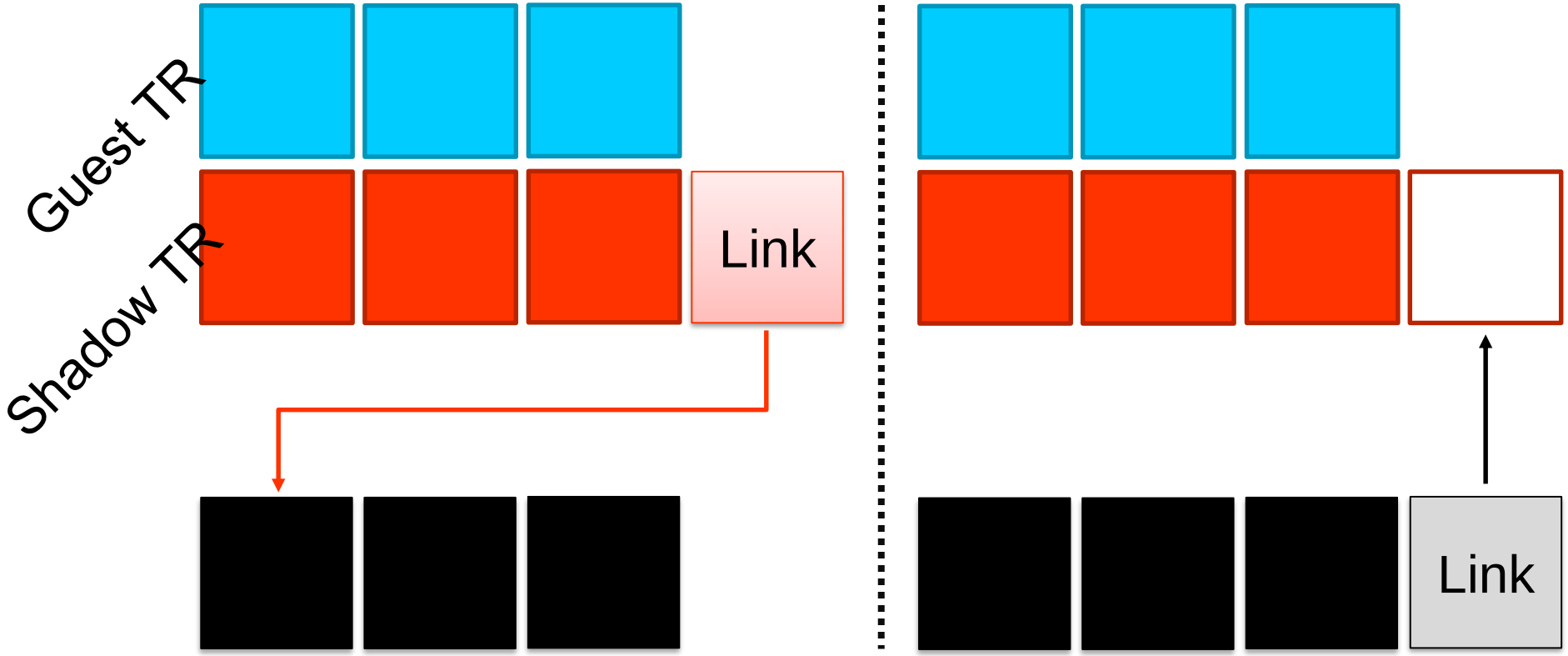
- If we put our TRBs into the shadow TR, it will make synchronization much harder





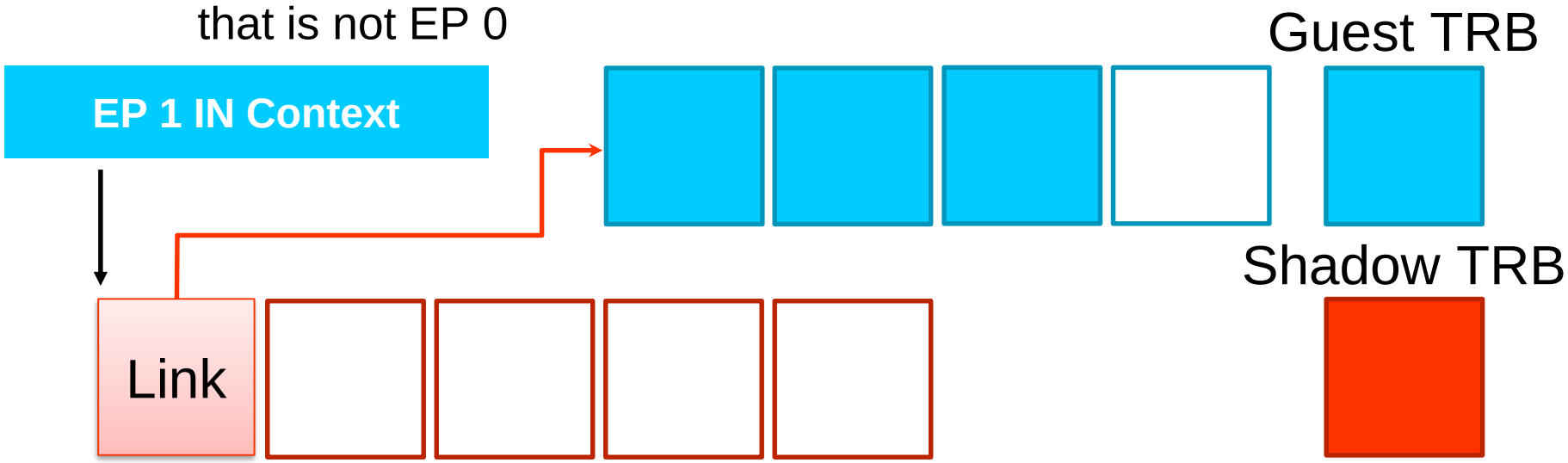
EP0 Ownership

- Jump to a dedicated TR owned BitVisor only, and jump back the original position



EP Ownership

- Not all USB drivers need shadowing
  - Let BitVisor owns all EPs at the beginning
  - A USB Driver must tell xHCI driver that it is going to own the device
  - If no USB driver claims the slot, put a Link TRB pointing to the guest Transfer Ring when the guest starts sending a request that is not EP 0



Changes in BitVisor (2)

- Changes in `struct usb_operations`
  - Implement `will_take_ctrl()`
    - It takes a USB host object, and a USB device object as its arguments
- USB Drivers
  - Need to explicitly tells *HCI drivers if it needs to take control a device by calling `usbhc->op->will_take_control()`

State Synchronization

- Two things to do
  - Device Context copyback
    - Both Slot Context and EP contexts
    - Replace the host pointer addresses with the guest ones
  - Event Ring copyback

State Synchronization

```
void
clone_er_trbs_to_guest(struct xhci_host *host)
{
    ...
    for (i = 0; i < host->usable_intrs; i++) {
        ...
        do {
            ...
            for (i_trb = start_idx; i_trb < n_trbs; i_trb++) {
                toggle = XHCI_TRB_GET_C(&h_trbs[i_trb]);
                if (toggle == current_toggle) {
                    handle_ev_trb(host, &h_trbs[i_trb]);
                } else {
                    ...
                }
            }
            ...
            memcpy(&g_trbs[start_idx], &h_trbs[start_idx],
                   (i_trb - start_idx + 1) * XHCI_TRB_NBYTES);
        } while (!stop);
    }
}
```

State Synchronization

```
static void
handle_ev_trb(struct xhci_host *host, struct xhci_trb *h_ev_trb)
{
    ...
    switch (type) {
        ...
        case XHCI_TRB_TYPE_TX_EV:
            slot_id = XHCI_EV_GET_SLOT_ID(h_ev_trb);
            ep_no   = XHCI_EV_GET_EP_NO(h_ev_trb);

            process_tx_ev(host, h_ev_trb);

            if (ep_no != 0 &&
                host->slot_meta[slot_id].host_ctrl == HOST_CTRL_NO) {
                break;
            }

            patch_tx_ev_trb(host, h_ev_trb);
            break;
        ...
    }
}
```

State Synchronization

```
static void
process_tx_ev(struct xhci_host *host, struct xhci_trb *h_ev_trb)
{
    ... /* Get h_urb list to find the target h_urb */
    ... /* Process all h_urbs that exist before */
    u8 is_last_intr = check_last_intr(h_urb, h_ev_trb, &trb_code);
    ...
    if (is_last_intr) {
        if (h_urb->shadow->buffers) {
            int res = usb_hook_process(host->usb_host,
                                       h_urb, USB_HOOK_REPLY);

            if (res == USB_HOOK_DISCARD) {
                ...
            }
        }

        remove_h_urb_head_from_ep(host, slot_id, ep_no);

        delete_urb_xhci(h_urb->shadow);
        delete_urb_xhci(h_urb);
    }
}
```

State Synchronization

- When to synchronize status?
  - It is possible to check for `OPR.USBSTS` access for events
  - Unfortunately, it is not a requirement. Linux writes to this register, but Windows occasionally reads it only
  - We have to intercept interrupts

Changes in BitVisor (3)

igel

- Intercepting interrupts
  - Every external interrupts cause VM_EXIT events
  - Implement `exint_pass_intr_register_callback()`
  - Implement `pci_register_intr_callback()`
    - It calls `exint_pass_intr_register_callback()` internally
  - xHCI driver calls `pci_register_intr_callback()` for registering its state synchronization callback

Summary

- xHCI Overview
  - Registers, data structures, and operational model
- Relationship between BitVisor's host controller drivers and USB Drivers
  - *HCI driver responsibilities
- xHCI Driver implementation
  - Implementation walkthrough
  - Changes in BitVisor