Managing Resources in FreeBSD Bus Drivers

John Baldwin

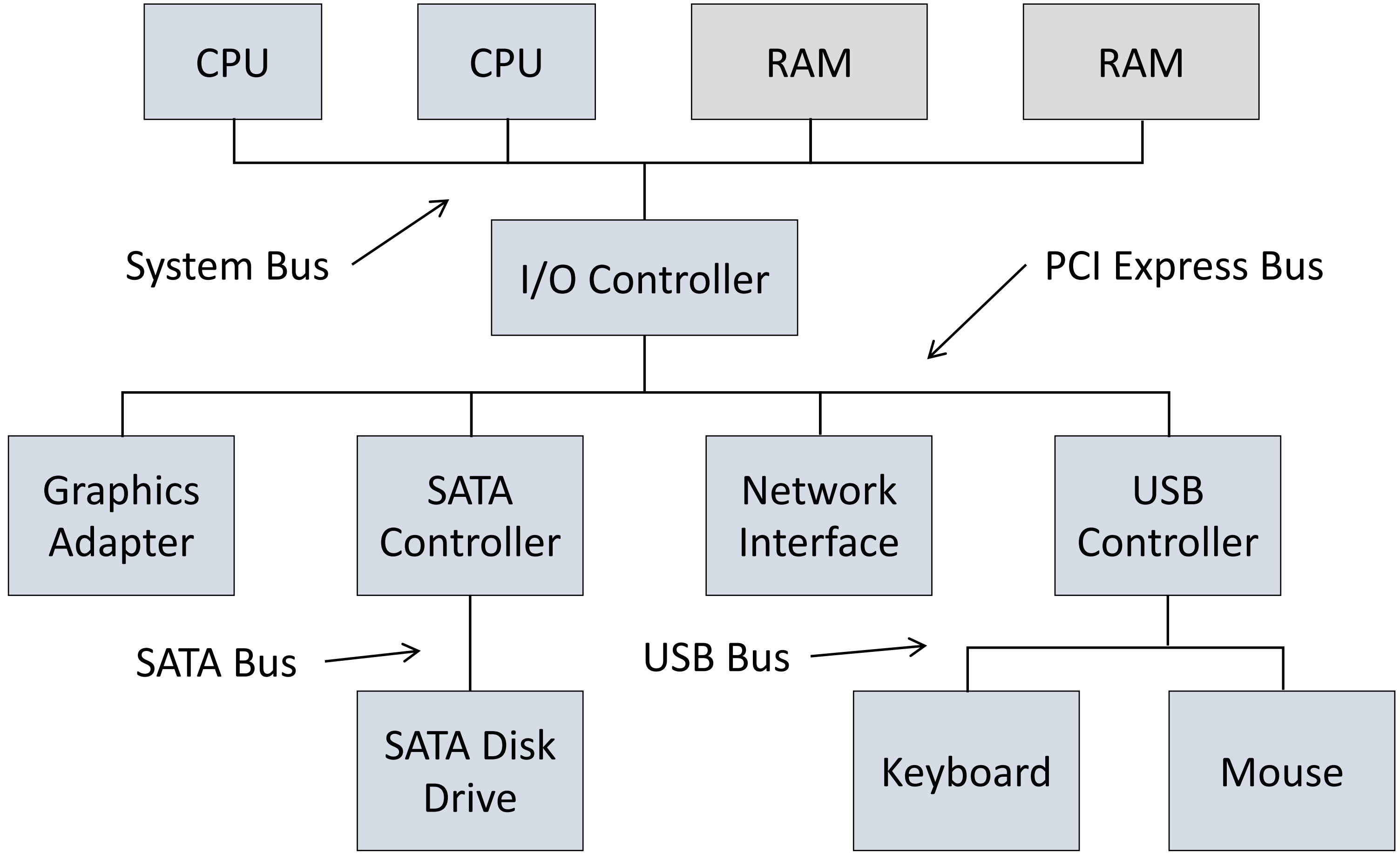
EuroBSDCon

22 September 2024

Background: Buses and Bridges

- Computer Systems contain multiple components that need to communicate
  - Processors, I/O Devices, Memory Controllers
- Components are connected to communication channels (“buses”)
- Bridges are a special type of device that are connected to two or more buses
- Bridges forward requests between components on different buses
  - Might need to translate messages
- Components organized in a hierarchy

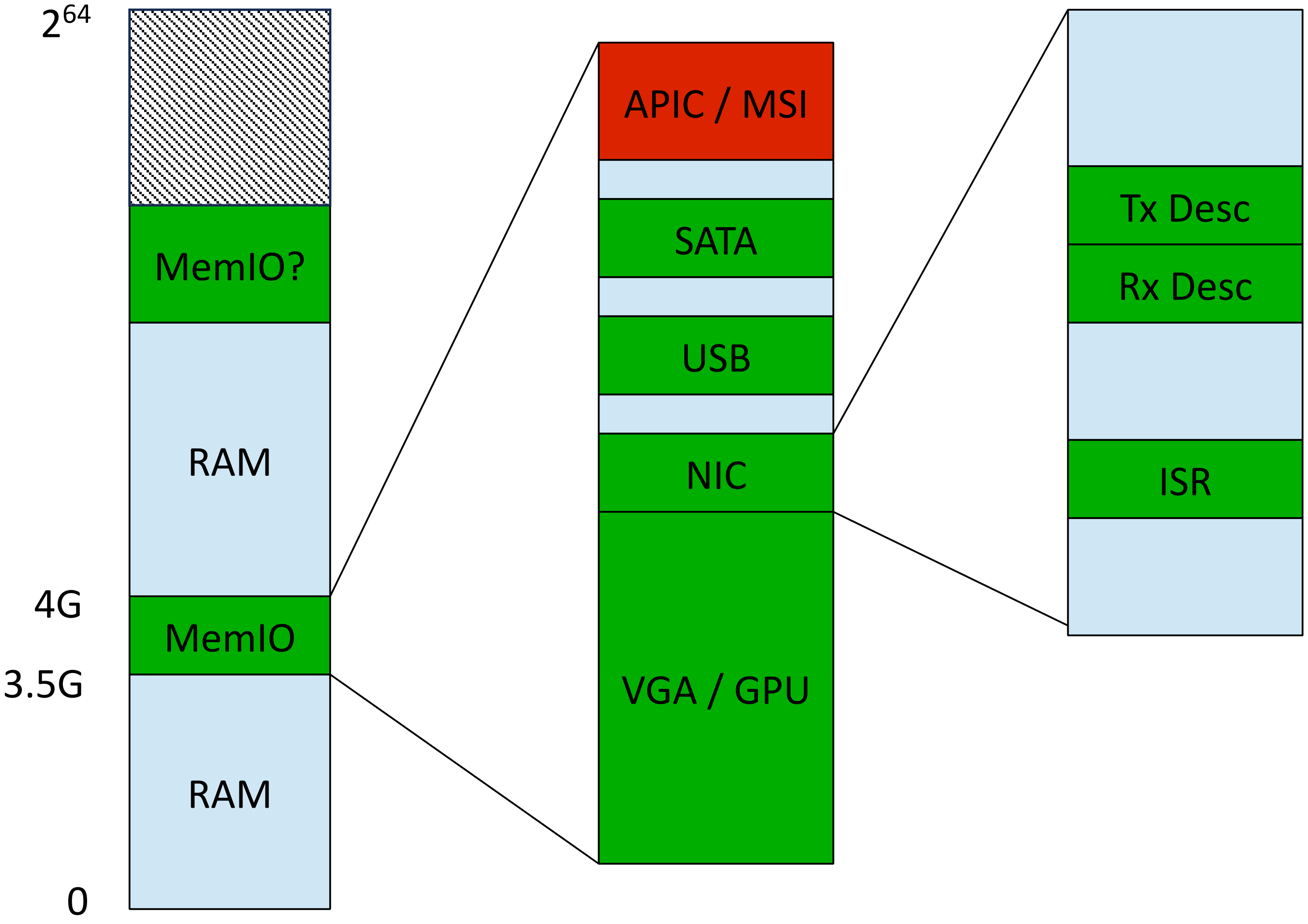
Device Hierarchy



Background: Device Resources

- Access to registers from application CPU (MMIO)
- Interrupting the CPU
  - “Where does my interrupt go?”

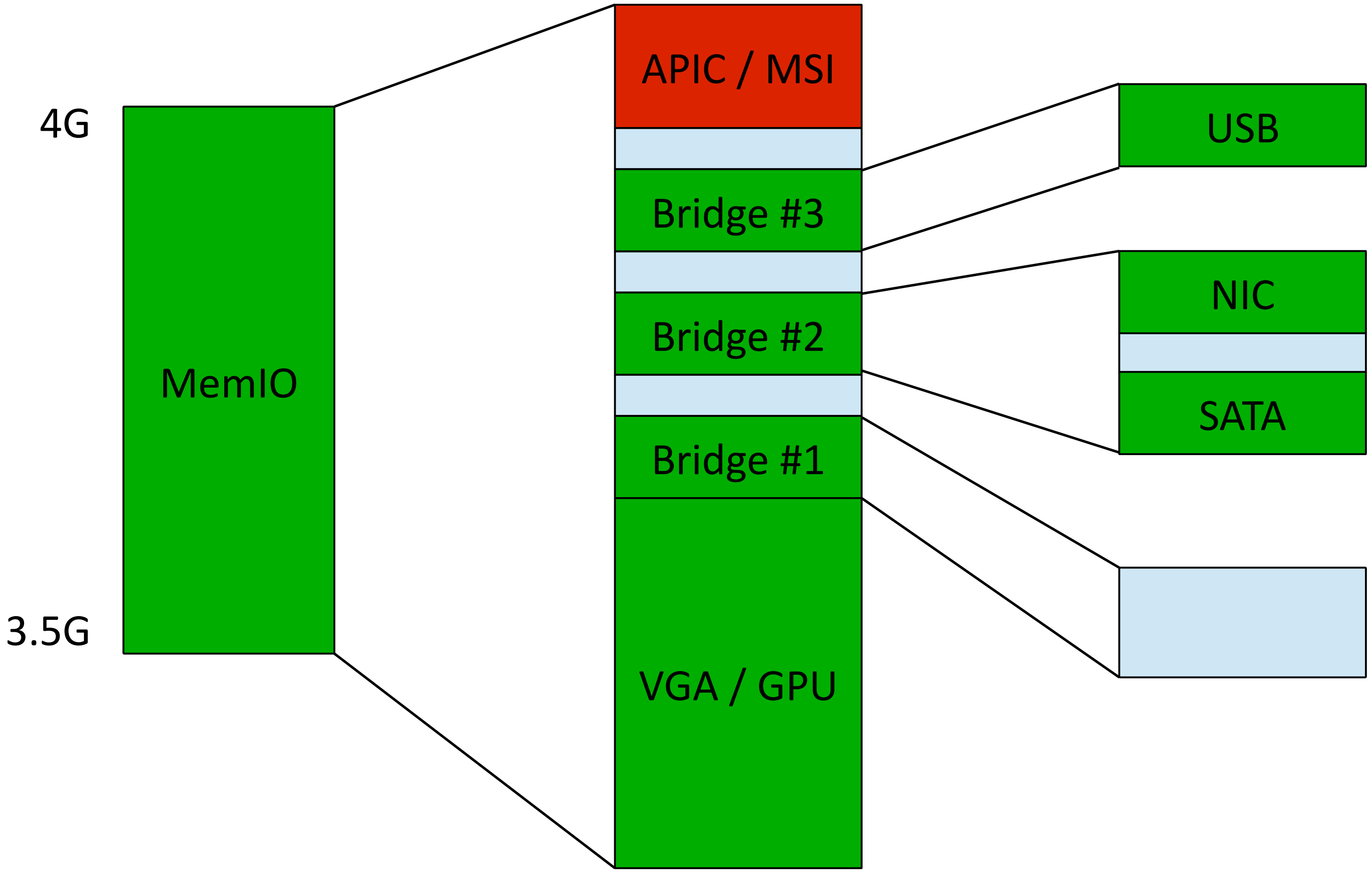
x86 Memory Address Space



Bridges and I/O Windows

- Bridges can use I/O windows to provide resources for child devices
- Windows claim a region of address space
- Child device resources are subsets of window address space
- Some bridges might translate resources
  - E.g. non-x86 top-level PCI bridges might map I/O ports to a range of MMIO

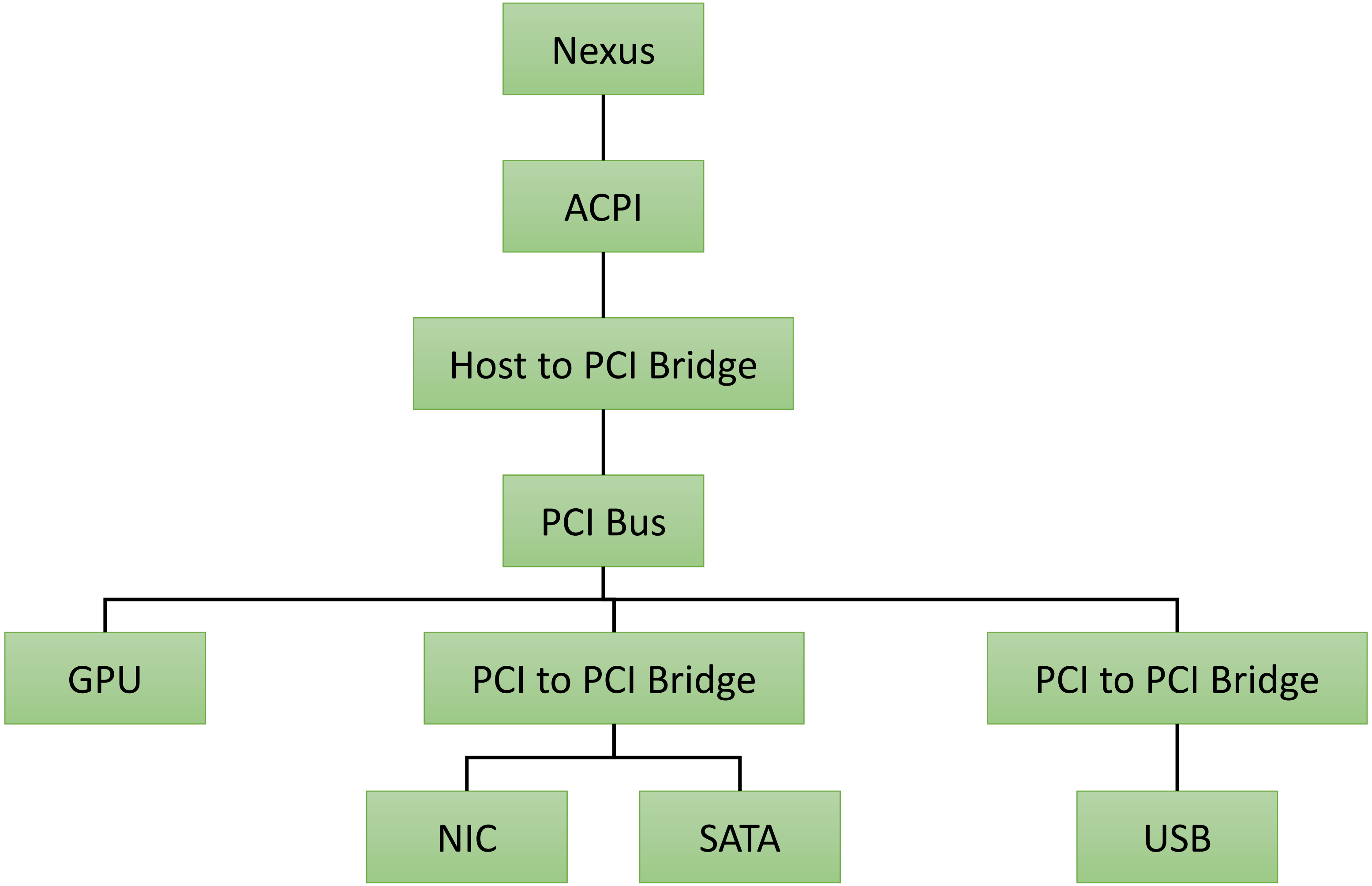
PCI Bridge Windows



Bus Devices in FreeBSD

- Any device that has child devices is a bus device
- Physical buses are a single device
  - PCI buses, ISA (LPC) buses
- Bridges are also bus devices
  - Child of a bus device
  - Another bus device as a child
- Pseudo buses
  - nexus0
  - ACPI, simplebus

FreeBSD Device Tree



Bus Drivers and Resources

- Bus drivers provide access to shared resources for children
- Must avoid resource conflicts
- Must provide valid resources
  - MMIO regions within the window
  - Interrupts device is connected to
- Must permit child devices to use allocated resources
  - Read/write registers via MMIO
  - Setup/teardown interrupt handlers

Resource Managers: `struct rman`

- Describe a subdividable resource via addressable ranges
- Effectively a special purpose address space manager
- Can describe either “real” address spaces (e.g. CPU physical) or virtual address spaces (IRQs for interrupt pins)
- Initialized with absolute bounds of address range and one or more available **regions** which can be sub-allocated
- Single allocated range described by `struct resource` object
- Avoids resource conflicts

Device Resources

- Resources belonging to devices are named by a tuple of **type** and resource ID (**rid**)
- Types include `SYS_RES_MEMORY`, `SYS_RES_IRQ`, `PCI_RES_BUS`
- Bus defines scheme for resource IDs
  - ACPI, simplebus use 0..N for each type
  - PCI uses `PCIR_BAR(x)` for BARs, 0 for INTx, 1…N for MSI/MSI-X

Resource Lists: `struct resource_list`

- Holds linked-list of `struct resource_list_entry` objects
- Each resource list entry (**rle**) contains resource type, rid, size, range
- Existence of a resource list entry does not allocate backing resources from the system (e.g. MMIO range)
- Backing resources are allocated from a resource manager, and the associated `struct resource` object is stored in the rle

Device Resource States

- Buses which can identify resources of children should **reserve** device resources after enumerating a device and keep them reserved as long as they are valid (even if a driver hasn't allocated them)
- Resource is **allocated** once a child device driver has requested it
- Allocated resources must be **activated** to use (read/write, add interrupt handler)
  - Typically via `RF_ACTIVE`
- Memory and I/O port resources must be **mapped** for use with `bus_read/write_*`

Resource Lists: API

- Bus driver should store resource list in per-device instance variables (ivars)
- `resource_list_init/free()` to setup and teardown
- Add new resources via `resource_list_add()`
- If bus knows about resource, reserve it via `resource_list_reserve()`
- `resource_list_alloc()` and `resource_list_release()` are helpers to use in `bus_alloc_resource` and `bus_release_resource` DEVMETHODs

Generic Helper Routines

- Multiple groups of routines suitable for use either as `DEVMETHOD`s directly, or can be used to as helpers to implement `DEVMETHOD`s
- `bus_generic_`<method>
- `bus_generic_rl_`<method>
- `bus_generic_rman_`<method>

Generic Bus Methods: `bus_generic_*`

- Generally speaking, passes request up to parent device
- Few exceptions / misnomers
- `bus_generic_probe()` invokes `device_identify DEVMETHOD` on all child drivers
  - **Not** a suitable `device_probe DEVMETHOD`, should be called from bus driver's attach routine
  - Should probably be renamed to `bus_identify_children()`

Generic Bus Methods: `bus_generic_*`

- `bus_generic_attach()` attaches drivers to child devices
  - Not sufficient as a standalone attach routine, child devices need to be added first
  - Should probably be renamed to `bus_attach_children()`
- `bus_generic_detach()` detaches drivers from child devices
  - Not sufficient as a standalone detach routine, child devices need to be deleted so they are cleaned up
  - Should probably be renamed to `bus_detach_children()`
  - Possibly reimplement as `bus_detach_children()` followed by `device_delete_children()`

Generic Bus Methods: `bus_generic_rl_*`

- Provides methods for child drivers to add/remove device resources
  - `bus_set_resource`, `bus_get_resource`, `bus_delete_resource`
  - Only needed if child drivers can add device resources
- Provide methods for allocating and releasing resources
  - `bus_alloc_resource`, `bus_release_resource`
  - Suitable if bus device does not sub-allocate from its own resource managers but depends on parent device to allocate resources described by a resource list entry
- Bus driver must implement `bus_get_resource_list DEVMETHOD`

Generic Bus Methods: `bus_generic_rman_*`

- Provide methods for managing resources allocated from resource managers
  - `bus_alloc_resource`, `bus_activate_resource`, `bus_adjust_resource`, `bus_deactivate_resource`, `bus_release_resource`
- Bus driver must implement `bus_get_rman` `DEVMETHOD`
- `bus_activate_resource` helper requires `bus_map_resource` `DEVMETHOD`
- `bus_deactivate_resource` helper requires `bus_unmap_resource` `DEVMETHOD`

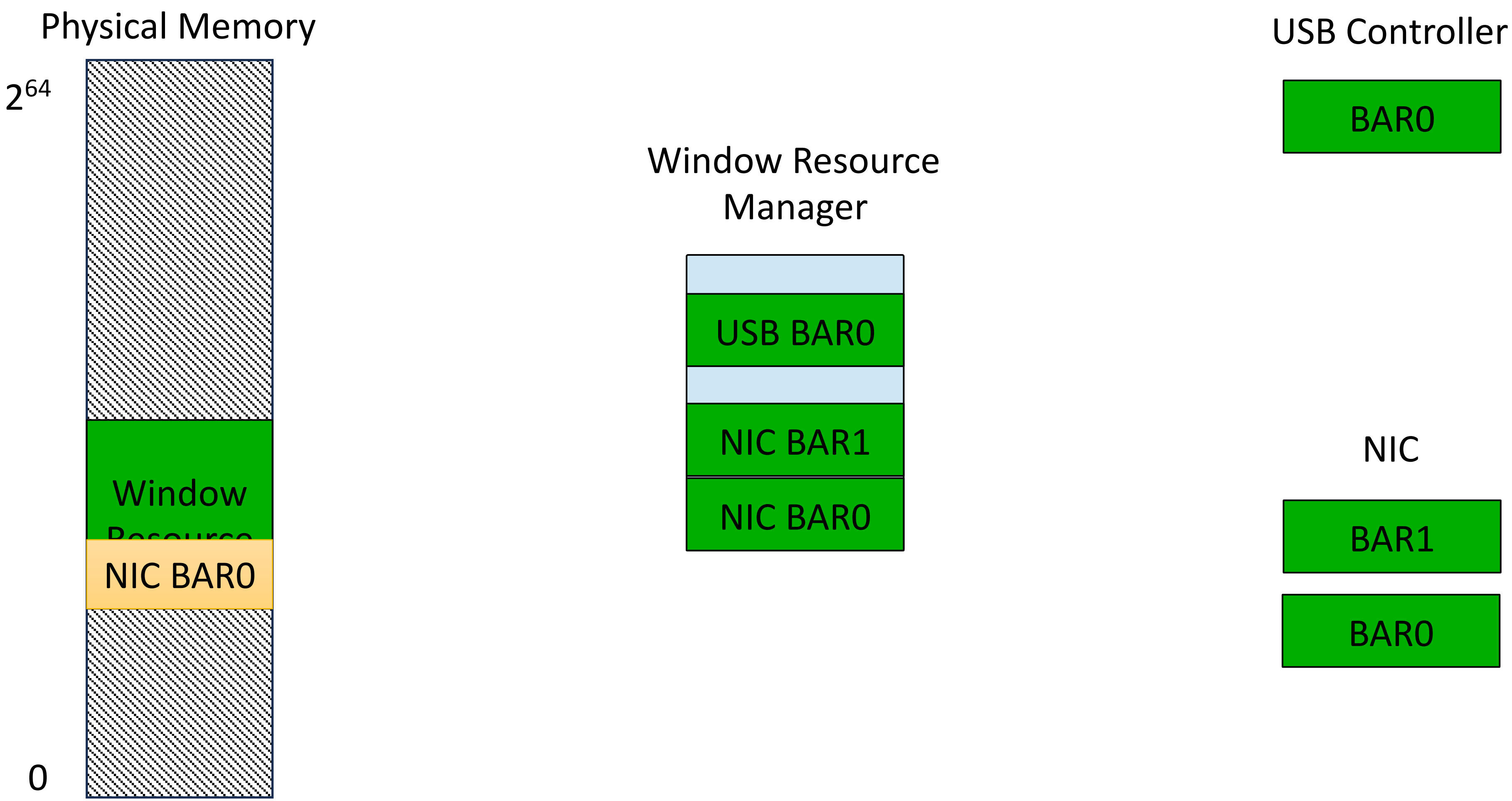
Example: PCI Bus Driver

- `pciX` bus devices represent a logical PCI bus with a parent bridge device (Host-PCI or PCI-PCI) and child PCI devices
- No resource managers, resources are provided by parent bridge device
  - Except for VFs which are a special case
- Resources for BARs are added while enumerating children during bus driver attach (`resource_list_add`)
- BAR resources are also reserved via `resource_list_reserve`
- Mostly uses `bus_generic_rl_*` and `resource_list_*`
- Custom activate/deactivate methods to deal with command register

Example: PCI-PCI Bridge Driver

- Bridges reserve resource ranges on parent bus via window config registers and sub-allocate from I/O windows for child devices
- Uses resource managers for each I/O window and `rman_*` to sub-allocate resources for child devices
- Allocates resource from parent bus for each I/O window
  - Resource is active (`RF_ACTIVE`) but unmapped (`RF_UNMAPPED`)
- Mapping requests for sub-allocated resources for child devices resolved by requesting a mapping of the suitable sub-range of the I/O window resource from the parent PCI bus

Example: PCI-PCI Bridge Driver



Example: nexus Driver

- `nexus0` is the root device of a system
- Uses resource managers for each global resource pool (e.g. all physical memory)
- Direct children must add any resources manually via `bus_set_resource`
- Uses `bus_generic_rl_*` for `bus_get/set/delete_resource`
- Uses `bus_generic_rman_*` for other bus resource methods
    - XXX: rle for direct children not updated

Example: Bridge Driver with Translation

- Some bridges translate resources (e.g. PCI memory address X becomes CPU physical X + N)
- Translated regions can be treated like a PCI-PCI bridge window
- Active but unmapped resource allocated from parent (e.g. using CPU physical address range) for each window
- Translated regions added to resource manager for each window
  - Resulting sub-allocated resources need to match what child device needs, e.g. values to write into PCI BARs
- Mapping a child device resource consists of mapping suitable sub-range of window resource allocated from parent

Peeking Under the Hood

- `devinfo -r` shows hierarchy of devices along with resources reserved by each device
- `devinfo -u` shows resource managers and allocations within each manager

Questions?