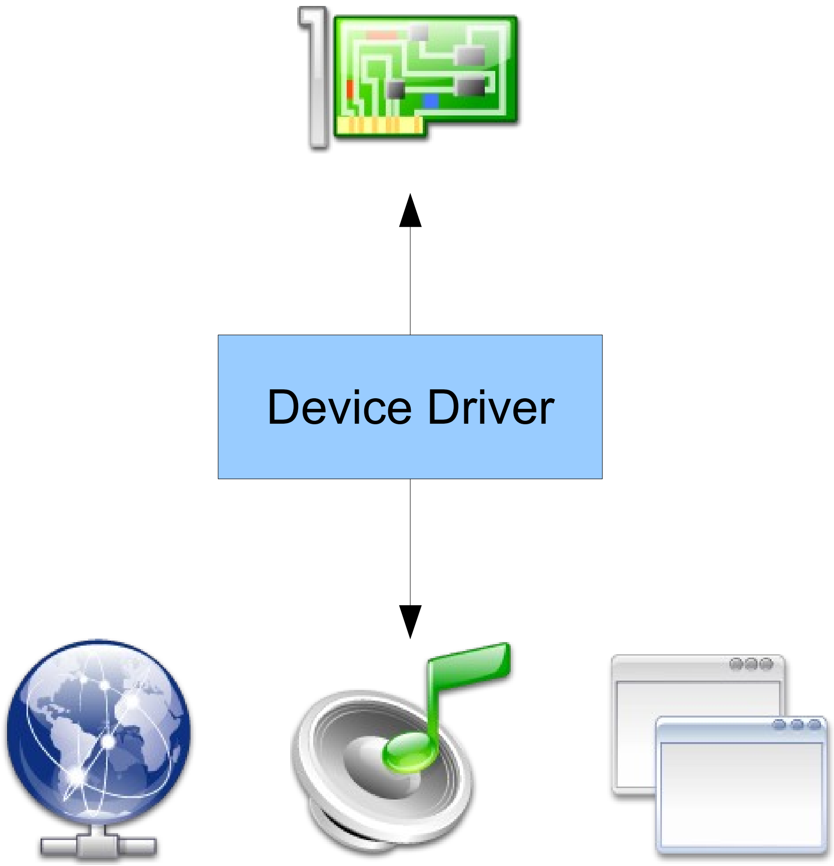
Writing and Adapting Device Drivers for FreeBSD

John Baldwin

November 5, 2011

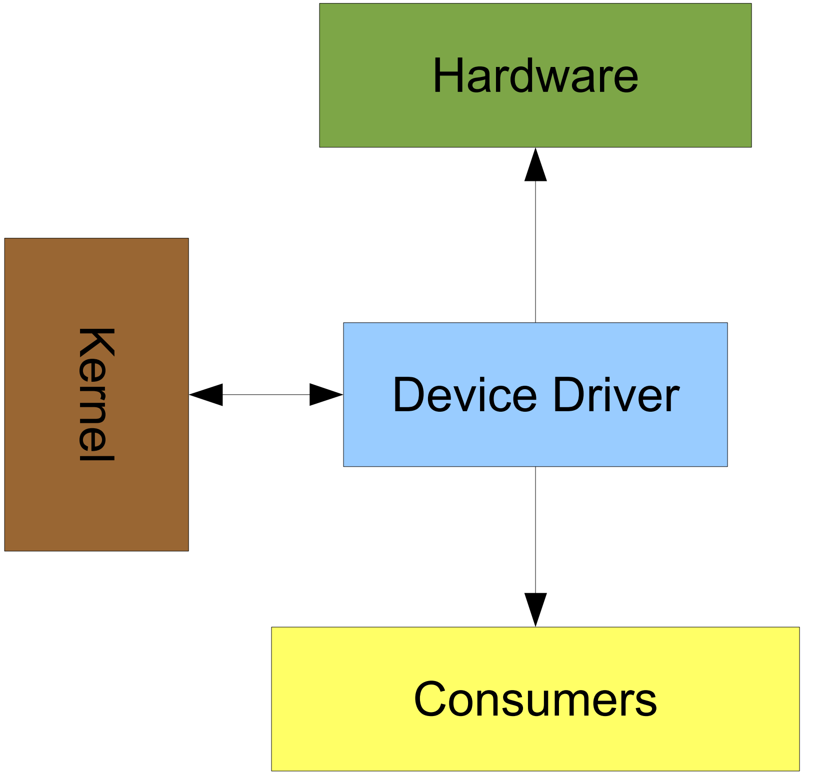
What is a Device Driver?

- Hardware
- Functionality
- A device driver is the software that bridges the two.



Focus of This Presentation

- In-kernel drivers for FreeBSD
- Drivers are built using various toolkits
  - Hardware
  - Kernel environment
  - Consumers
- ACPI and PCI



Roadmap

- Hardware Toolkits
  - Device discovery and driver life cycle
  - I/O Resources
  - DMA
- Consumer Toolkits
  - Character devices
  - ifnet(9)
  - disk(9)

Device Discovery and Driver Life Cycle

- New-bus devices
- New-bus drivers
- Device probe and attach
- Device detach

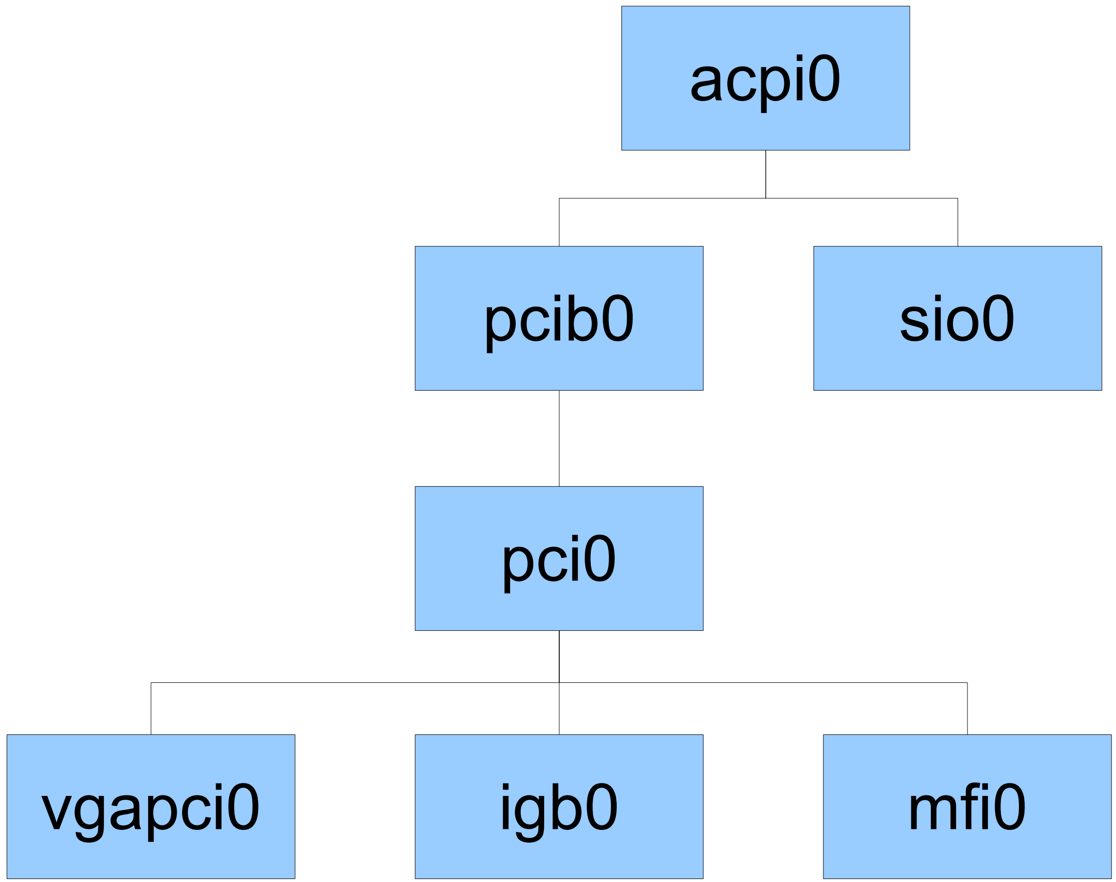
New-bus Devices

- `device_t` objects
  - Represent physical devices or buses
  - Populated by bus driver for self-enumerating buses (e.g. ACPI and PCI)
- Device instance variables (ivars)
  - Bus-specific state
  - Bus driver provides accessors
    - `pci_get_vendor()`, `pci_get_device()`
    - `acpi_get_handle()`

New-bus Drivers

- `driver_t` objects
  - Method table
  - Parent bus by name
  - Size of softc
- softc == driver per-instance state
  - Managed by new-bus framework
  - Allocated and zeroed at attach
  - Freed at detach

New-bus Device Tree



Device Probe and Attach

- Bus driver initiates device probes
  - Device arrival, either at boot or hotplug
  - Rescans when new drivers are added via `kldload(2)`
- `device_probe` method called for all drivers associated with the parent bus
- Winning driver is chosen and its `device_attach` method is called

Device Probe Methods

- Usually use ivars
- May poke hardware directly (rarely)
- Return value used to pick winning driver
  - Returns errno value on failure (typically `ENXIO`)
  - `device_set_desc()` on success
  - Values <= 0 indicate success
    - `BUS_PROBE_GENERIC`
    - `BUS_PROBE_DEFAULT`
    - `BUS_PROBE_SPECIFIC`
      - Special softc behavior!

Device Attach Methods

- Initialize per-device driver state (softc)
- Allocate device resources
- Initialize hardware
- Attach to Consumer Toolkits
- Returns 0 on success, errno value on failure
  - Must cleanup any partial state on failure

Device Detach

- Initiated by bus driver
  - Removal of hotplug device
  - Driver removal via `kldunload(2)`
- `device_detach` method called (“attach in reverse”)
  - Should detach from Consumer Toolkits
  - Quiesce hardware
  - Release device resources

Example 1: ipmi(4)

- ACPI and PCI attachments for ipmi(4)
- Method tables
- Probe routines
- `sys/dev/ipmi/ipmi_acpi.c`
- `sys/dev/ipmi/ipmi_pci.c`

Roadmap

- Hardware Toolkits
  - Device discovery and driver life cycle
  - I/O Resources
  - DMA
- Consumer Toolkits
  - Character devices
  - ifnet(9)
  - disk(9)

I/O Resources

- Resource Objects
- Allocating and Releasing Resources
- Accessing Device Registers
- Interrupt Handlers

Resource Objects

- Resources represented by `struct resource`
- Opaque and generally used as a handle
- Can access details via `rman(9)` API
  - `rman_get_start()`
  - `rman_get_size()`
  - `rman_get_end()`

Allocating Resources

- Parent bus driver provides resources
- `bus_alloc_resource()` returns pointer to a resource object
  - If bus knows start and size (or can set them), use `bus_alloc_resource_any()` instead
  - Typically called from device attach routine
- Individual resources identified by bus-specific resource IDs (`rid` parameter) and type
  - Type is one of `SYS_RES_*`

Resource IDs

- ACPI
  - 0..N based on order in `_CRS`
  - Separate 0..N for each type
- PCI
  - Memory and I/O port use `PCIR_BAR(x)`
  - INTx IRQ uses rid 0
  - MSI/MSI-X IRQs use rids 1..N

Releasing Resources

- Resources released via `bus_release_resource()`
- Typically called from device detach routine
- Driver responsible for freeing all resources during detach!

Detour: bus_space(9)

- Low-level API to access device registers
  - API is MI, implementation is MD
- A block of registers are described by a tag and handle
  - Tag typically describes an address space (e.g. memory vs I/O ports)
  - Handle identifies a specific register block within the address space
- Lots of access methods

Accessing Device Registers

- Resource object must be activated
  - Usually by passing the `RF_ACTIVE` flag to `bus_alloc_resource()`
  - Can use `bus_activate_resource()`
- Activated resource has a valid bus space tag and handle for the register block it describes
- Wrappers for bus space API
  - Pass resource instead of tag and handle
  - Remove “`_space`” from method name

Wrapper API Examples

- bus_read_<size>(resource, offset)
  - Reads a single register of size bytes and returns value
  - Offset is relative to start of resource
- bus_write_<size>(resource, offset, value)
  - Writes value to a single register of size bytes
  - Offset is relative to start of resource

Interrupt Handlers

- Two types of interrupt handlers: filters and threaded handlers
- Most devices will just use threaded handlers
- Both routines accept a single shared void pointer argument. Typically this is a pointer to the driver's softc.

Interrupt Filters

- Run in “primary interrupt context”
  - Use interrupted thread's context
  - Interrupts at least partially disabled in CPU
- Limited functionality
  - Only spin locks
  - “Fast” taskqueues
  - `swi_sched()`, `wakeup()`, `wakeup_one()`

Interrupt Filters

- Returns one of three constants
  - `FILTER_STRAY`
  - `FILTER_HANDLED`
  - `FILTER_SCHEDULE_THREAD`
- Primary uses
  - UARTs and timers
  - Shared interrupts (not common)
  - Workaround broken hardware (em(4) vs Intel PCH)

Threaded Handlers

- Run in a dedicated interrupt thread
  - Dedicated context enables use of regular mutexes and rwlocks
  - Interrupts are enabled
- Greater functionality
  - Anything that doesn't sleep
  - Should still defer heavyweight tasks to a taskqueue
- No return value

Attaching Interrupt Handlers

- Attached to `SYS_RES_IRQ` resources via `bus_setup_intr()`
- Can register a filter, threaded handler, or both
- Single void pointer arg passed to both filter and threaded handler

Attaching Interrupt Handlers

- Flags argument to `bus_setup_intr()` must include one of `INTR_TYPE_*`
- Optional flags
  - `INTR_ENTROPY`
  - `INTR_MPSAFE`
- A void pointer cookie is returned via last argument

Detaching Interrupt Handlers

- Pass `SYS_RES_IRQ` resource and cookie to `bus_teardown_intr()`
- Ensures interrupt handler is not running and will not be scheduled before returning
- May sleep

Example 2: ipmi(4)

- ACPI and PCI resource allocation for ipmi(4)
- Attach routines
- `sys/dev/ipmi/ipmi_acpi.c`
- `sys/dev/ipmi/ipmi_pci.c`

Example 2: ipmi(4)

- Accessing device registers
  - `INB()` and `OUTB()` in `sys/dev/ipmi/ipmivars.h`
  - `sys/dev/ipmi/ipmi_kcs.c`
- Configuring interrupt handler
  - `sys/dev/ipmi/ipmi.c`

Roadmap

- Hardware Toolkits
  - Device discovery and driver life cycle
  - I/O Resources
  - DMA
- Consumer Toolkits
  - Character devices
  - ifnet(9)
  - disk(9)

DMA

- Basic concepts
- Static vs dynamic mappings
- Deferred callbacks
- Callback routines
- Buffer synchronization

bus_dma(9) Concepts

- `bus_dma_tag_t`
  - Describes a DMA engine's capabilities and limitations
  - Single engine may require multiple tags
- `bus_dmamap_t`
  - Represents a mapping of a single I/O buffer
  - Mapping only active while buffer is “loaded”
  - Can be reused, but only one buffer at a time

Static DMA Mappings

- Used for fixed allocations like descriptor rings
- Size specified in tag, so usually have to create dedicated tags
- Allocated via `bus_dmamem_alloc()` which allocates both a buffer and a DMA map
- Buffer and map must be explicitly loaded and unloaded
- Released via `bus_dmamem_free()`

Dynamic DMA Mappings

- Used for I/O buffers (`struct bio`, `struct mbuf`, `struct uio`)
- Driver typically preallocates DMA maps (e.g. one for each entry in a descriptor ring)
- Map is bound to I/O buffer for life of transaction via `bus_dmamap_load*()` and `bus_dmamap_unload()` and is typically reused for subsequent transactions

Deferred Callbacks

- Some mapping requests may need bounce pages
- Sometimes there will be insufficient bounce pages available
- Driver is typically running in a context where sleeping would be bad
- Instead, if caller does not specify `BUS_DMA_NOWAIT`, the request is queued and completed asychronously

Implications of Deferred Callbacks

- Cannot assume load operation has completed after `bus_dmamap_load()` returns
- If request is deferred, `bus_dmamap_load()` returns `EINPROGRESS`
- To preserve existing request order, driver is responsible for “freezing” its own request queue when a request is deferred
  - bus_dma(9) lies, all future requests are not queued automatically

Non-Deferred Callbacks

- Can pass `BUS_DMA_NOWAIT` flag in which case `bus_dmamap_load()` fails with `ENOMEM` instead
- `bus_dmamap_load_mbuf()`, `bus_dmamap_load_mbuf_sg()`, and `bus_dmamap_load_uio()` all imply `BUS_DMA_NOWAIT`
- Static mappings will not block and should use `BUS_DMA_NOWAIT`

Callback Routines

- When a load operation succeeds, the result is passed to the callback routine
- Callback routine is passed a scatter/gather list and an error value
- If scatter/gather list would contain too many elements, `EFBIG` error is passed to callback routine (not returned from `bus_dmamap_load*()`)
  - Bounce pages not used to defrag automatically

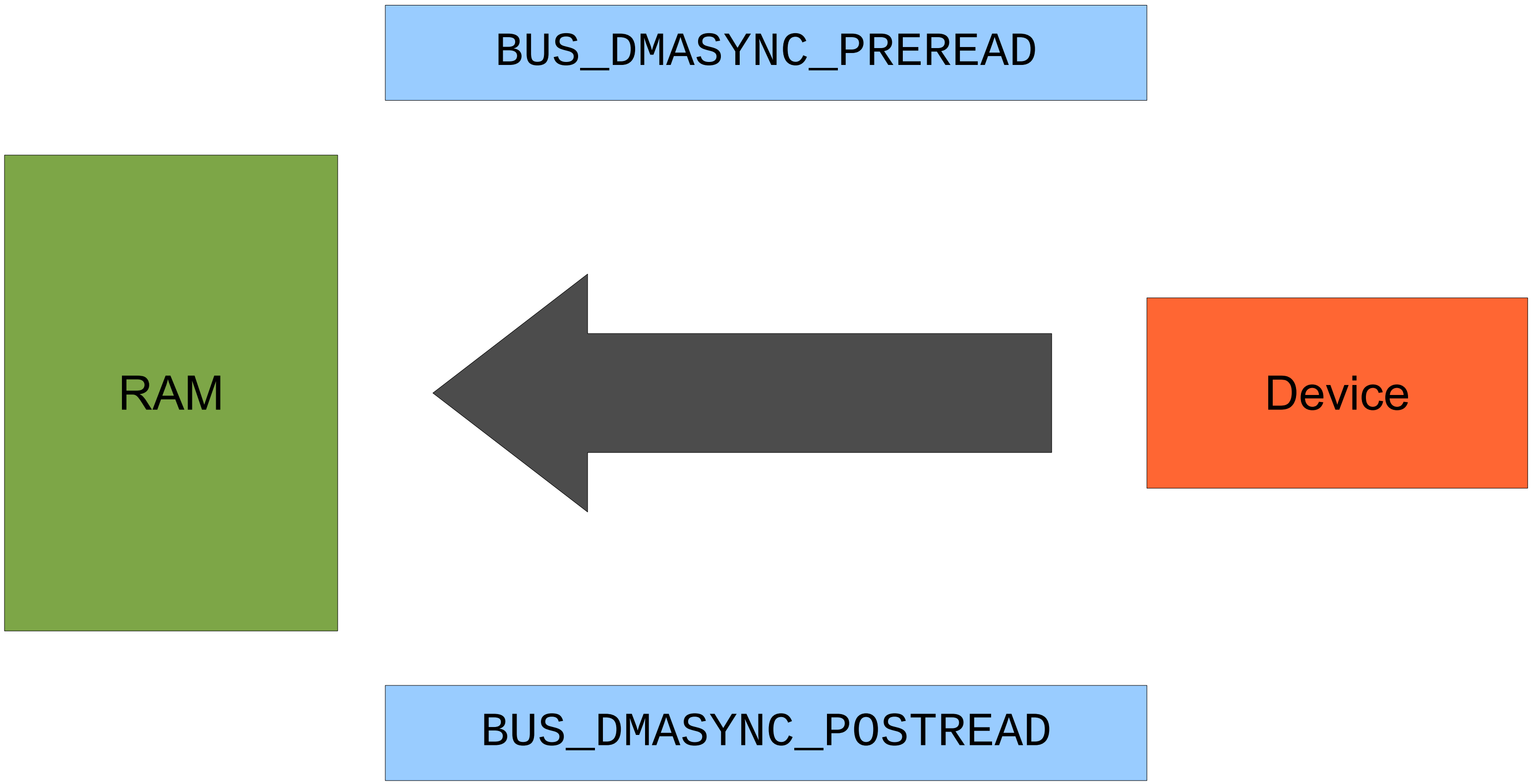
`bus_dmamap_load_mbuf_sg()`

- More convenient interface for NIC drivers
- Caller provides S/G list (and must ensure it is large enough)
- No callback routine, instead it will return `EFBIG` directly to the caller
- Typical handling of `EFBIG`
  - `m_collapse()` first (cheaper)
  - `m_defrag()` as last resort

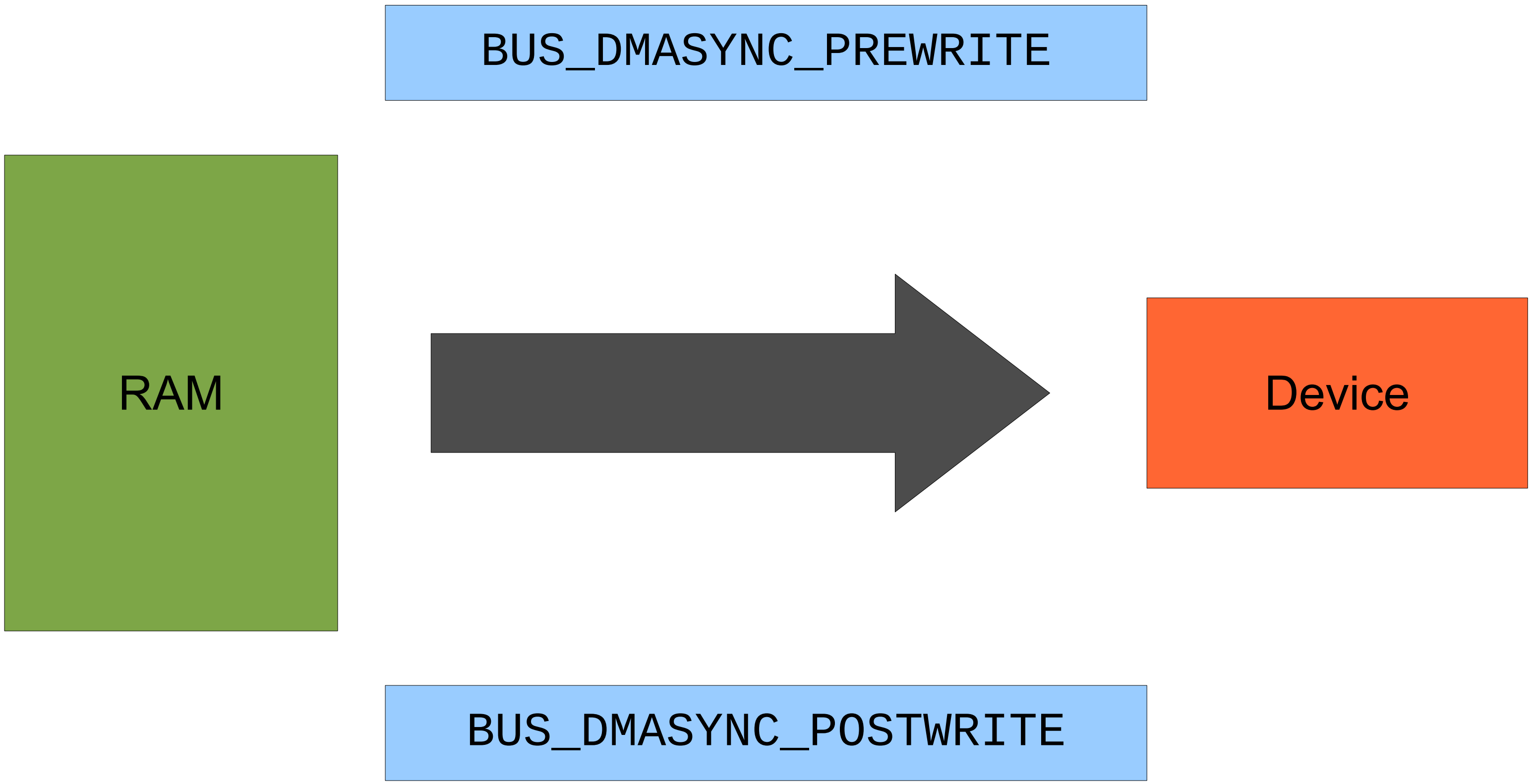
Buffer Synchronization

- `bus_dmamap_sync()` is used to ensure CPU and DMA mappings are in sync
  - Memory barriers
  - Cache flushes
  - Bounce page copies
- Operates on loaded map
- The `READ`/`WRITE` field in operations are with respect to CPU, not device

Buffer Synchronization: READ



Buffer Synchronization: WRITE



Example 3: de(4)

- `sys/dev/de/if_de.c`
- Static allocation for descriptor rings
  - `tulip_busdma_allocring()`
- Dynamic allocation for mbufs
  - Tag and maps created in `tulip_busdma_allocring()`
  - Mapping TX packet in `tulip_txput()`

Example 4: mfi(4)

- `sys/dev/mfi/mfi.c`
- `mfi_mapcmd()` and `mfi_data_cb()` queue DMA requests to controller
- `mfi_intr()` unfreezes queue when pending requests complete

Roadmap

- Hardware Toolkits
  - Device discovery and driver life cycle
  - I/O Resources
  - DMA
- Consumer Toolkits
  - Character devices
  - ifnet(9)
  - disk(9)

Character Devices

- Data structures
- Construction and destruction
- Open and close
- Basic I/O
- Event notification
- Memory mapping
- Per-open file descriptor data

Data Structures

- `struct cdevsw`
  - Method table
  - Flags
  - Set version to `D_VERSION`
- `struct cdev`
  - Per-instance data
  - Driver fields
    - `si_drv1` (typically softc)
    - `si_drv2`

Construction and Destruction

- `make_dev()`
  - Creates cdev or returns existing cdev
  - Uses passed in cdevsw when creating cdev
  - Drivers typically set `si_drv1` in the returned cdev after construction
- `destroy_dev()`
  - Removes cdev
  - Blocks until all threads drain
    - `d_purge()` cdevsw method

Open and Close

- The `d_open()` method is called on every `open()` call
  - Permission checks
  - Enforce exclusive access
- The `d_close()` method is called only on the last `close()` by default
- The `D_TRACKCLOSE` flag causes `d_close()` for each `close()`

Caveats of Close

- `d_close()` may be invoked from a different thread or process than `d_open()`
- `D_TRACKCLOSE` can miss closes if a devfs mount is force-unmounted
  - `cdevpriv(9)` is a more robust alternative

    (more on that later)

Basic I/O

- d_read() and d_write()
  - struct uio provides request details
    - uio_offset is desired offset
    - uio_resid is total length
  - uiomove(9) copies data between KVM and uio
  - ioflag holds flags from <sys/vnode.h>
    - IO_NDELAY (O_NONBLOCK)
    - IO_DIRECT (O_DIRECT)
    - IO_SYNC (O_FSYNC)
      - fcntl(F_SETFL) triggers FIONBIO and FIOASYNC ioctls

Basic I/O

- `d_ioctl()`
  - `cmd` is an `ioctl()` command (`_IO()`, `_IOR()`, `_IOW()`, `_IOWR()`)
    - Read/write is from requester's perspective
  - `data` is a kernel address
    - Kernel manages copyin/copyout of data structure specified in ioctl command
  - `fflag`
    - `O_*` flags from `open()` and `FREAD` and `FWRITE`
    - No implicit read/write permission checks!

Event Notification

- Two frameworks to signal events
- `select()` / `poll()`
  - Only `read()` and `write()`
- `kevent()`
  - Can do `read()` / `write()` as well as custom filters
- Driver can support none, one, or both
  - `select()` / `poll()` will always succeed if not implemented
  - `kevent()` will fail to attach event

select() and poll()

- Need a struct selinfo to manage sleeping threads
  - seldrain() during device destruction
- d_poll()
  - POLL* constants in <sys/poll.h>
  - Returns a bitmask of requested events that are true
  - If no events to return and requested events includes relevant events, call selrecord()
- When events become true, call selwakeup()

kevent()

- Need a knote list to track active knotes
  - struct selinfo includes a note in si_note
  - knlist_init*() during device creation
  - knlist_destroy() during device destruction
- Each filter needs a struct filterops
  - f_isfd should be 1
  - f_attach should be NULL
  - Attach done by d_kqfilter() instead

Filter Operations

- d_kqfilter()
  - Assign struct filterops to kn_ops
  - Set cookie in kn_hook (usually softc)
  - Add knote to knote list via knlist_add()
- f_event()
  - Set kn_data and kn_fflags
  - Return true if event should post
- f_detach()
  - Remove knote from list via knlist_remove()

KNOTE()

- Signals that an event should be posted to a list
- `f_event()` of all knotes on list is called
  - Each knote determines if it should post on its own
- `hint` argument is passed from `KNOTE()` to each `f_event()`

Knote Lists and Locking

- Knote list operations are protected by a global mutex by default
- Can re-use your own mutex if desired
  - Pass as argument to `knlist_init_mtx()`
- Use *_locked variants of `KNOTE()` and knlist operations if lock is already held
- `f_event()` will always be called with lock already held

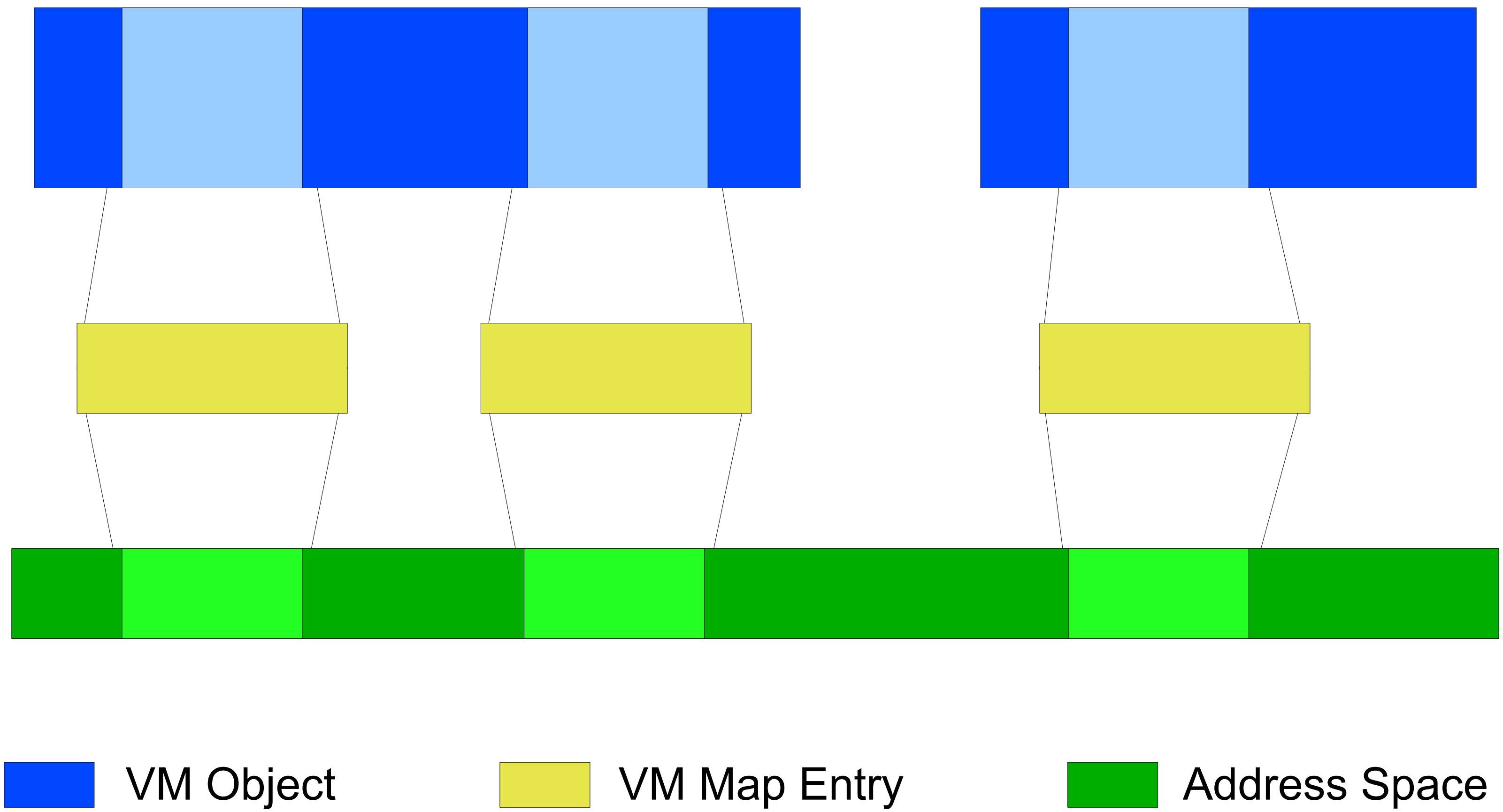
Example 5: echodev(4)

- http://www.freebsd.org/~jhb/papers/drivers/echodev
- `/dev/echobuf`
  - Addressable, variable-sized buffer
  - Readable and writable as long as buffer has non-zero size
- `/dev/echostream`
  - Stream buffer, so ignores `uio_offset`
  - Readable and writable semantics like a TTY or pipe

Memory Mapping

- VM objects (`vm_object_t`) represent something that can be mapped and define their own address space using pager methods
  - Files (vnode pager)
  - Anonymous objects (default pager)
  - Devices (device pager)
- An address space (`struct vmspace`) contains a list of VM map entries each of which maps a portion of an object's address space

Memory Mapping



Device Pager

- Each character device has exactly one device pager VM object
- Object's address space is defined by `d_mmap()` method
- Object's address space is static, once a mapping is established for a page it lives forever
- `close()` does not revoke mappings
  - `destroy_dev()` does not invalidate object(!)

d_mmap()

- Returns zero on success, error on failure
- Object offset will be page aligned
- Returned *paddr must be page aligned
- Desired protection is mask of PROT_*
- May optionally set *memattr to one of VM_MEMATTR_*
    - Defaults to VM_MEMATTR_DEFAULT

d_mmap() Invocations

- Called for each page to check permissions on each mmap()
  - Uses protection from mmap() call
- Called on first page fault for each object page
  - Uses PROT_READ for protection
  - Must not fail, results cached forever
  - Invoked from arbitrary thread
    - No per-open file descriptor data (cdevpriv)

d_mmap_single()

- Called once per mmap() with entire length, not per-page
- Can return ENODEV to fallback to device pager
- May optionally supply arbitrary VM object to satisfy request by returning zero
  - Can use any of offset, size, and protection as key
  - Must obtain reference on returned VM object
  - May modify offset (it is relative to returned object)

Per-open File Descriptor Data

- Can associate a void pointer with each open file descriptor
- A driver-supplied destructor is called when the file descriptor's reference count drops to zero
  - Typically contains logic previously done in `close()`
- Can be fetched from any cdevsw routine except for `d_mmap()` during a page fault

cdevpriv API

- `devfs_set_cdevpriv()`
  - Associates void pointer and destructor with current file descriptor
  - Will fail if descriptor already has associated data
- `devfs_get_cdevpriv()`
  - Current data is returned via `*datap`
  - Will fail if descriptor has no associated data
- `devfs_clear_cdevpriv()`
  - Clears associated data and invokes destructor

Example 6: lapicdev(4) & memfd(4)

- http://www.freebsd.org/~jhb/papers/drivers/lapicdev
- `/dev/lapic`
  - Maps the local APIC uncacheable and read-only using `d_mmap()`
- http://www.freebsd.org/~jhb/papers/drivers/memfd
- `/dev/memfd`
  - Creates swap-backed anonymous memory for each open file descriptor
  - Uses cdevpriv and `d_mmap_single()`

Roadmap

- Hardware Toolkits
  - Device discovery and driver life cycle
  - I/O Resources
  - DMA
- Consumer Toolkits
  - Character devices
  - ifnet(9)
  - disk(9)

Network Interfaces

- `struct ifnet`
- Construction and Destruction
- Initialization and Control
- Transmit
- Receive

struct ifnet

- `if_softc` typically used by driver to point at softc
- Various function pointers, some set by driver and others by link layer
- `if_flags` and `if_drv_flags` hold `IFF_*` flags
- Various counters such as `if_ierrors`, `if_opackets`, and `if_collisions`

Construction

- Allocated via `if_alloc(IFT_*)` (typically `IFT_ETHER`) during device attach
- `if_initname()` sets interface name, often reuses `device_t` name
- Driver should set `if_softc`, `if_flags`, `if_capabilities`, and function pointers
- `ether_ifattach()` called at end of device attach to set link layer properties

Destruction

- `ether_ifdetach()` called at beginning of device detach
- Device hardware should be shutdown after `ether_ifdetach()` to avoid races with detach code invoking `if_ioctl()`
- `if_free()` called near end of device detach when all other references are removed

if_init()

- Invoked when an interface is implicitly marked up (IFF_UP) when an address is assigned
- Commonly reused in if_ioctl() handlers when IFF_UP is toggled
- Should enable transmit and receive operation and set IFF_DRV_RUNNING on success
- Sole argument is value of if_softc
- Drivers typically include a “stop” routine as well

if_ioctl()

- Used for various control operations
  - SIOCSIFMTU (if jumbo frames supported)
  - SIOCSIFFLAGS
    - IFF_UP
    - IFF_ALLMULTI and IFF_PROMISC
  - SIOCADDMULTI / SIOCDELMULTI
  - SIOCIFCAP (IFCAP_* flags)
- Should use ether_ioctl() for the default case

Transmit

- Network stack provides Ethernet packets via `struct mbuf` pointers
- Driver responsible for free'ing mbufs after transmit via `m_freem()`
- Driver passes mbuf to `BPF_MTAP()`
- Two transmit interfaces
  - Traditional interface uses stack-provided queue
  - Newer interface dispatches each packet directly to driver

IFQUEUE and if_start()

- Network stack queues outbound packets to an interface queue (initialized during attach)
- Stack invokes if_start() method if IFF_DRV_OACTIVE is clear
- if_start() method drains packets from queue using IFQ_DRV_DEQUEUE(), sets IFF_DRV_OACTIVE if out of descriptors
- Interrupt handler clears IFF_DRV_OACTIVE and invokes if_start() after TX completions

if_transmit() and if_qflush()

- Driver maintains its own queue(s)
- Network stack always passes each packet to `if_transmit()` routine
- `if_transmit()` routine queues packet if no room
- Interrupt handler should transmit queued packets after handling TX completions
- Network stack invokes `if_qflush()` to free queued packets when downing interface

Receive

- Driver pre-allocates mbufs to receive packets
- Interrupt handler passes mbufs for completed packets up stack via if_input()
  - Must set lengths and received interface
  - Can also set flow id (RSS), VLAN, checksum flags
  - Cannot hold any locks used in transmit across `if_input()` call
  - Should replenish mbufs on receive

Example 7: xl(4)

- `sys/dev/xl/if_xl.c`
- `struct ifnet` allocation and IFQ setup in `xl_attach()`
- Control request handling in `xl_ioctl()`
- Transmitting IFQ in `xl_start_locked()`
- Received packet handling in `xl_rxeof()`
- Transmit completions in `xl_txeof()` and `xl_intr()`

Roadmap

- Hardware Toolkits
  - Device discovery and driver life cycle
  - I/O Resources
  - DMA
- Consumer Toolkits
  - Character devices
  - ifnet(9)
  - disk(9)

Disk Devices

- I/O operations – `struct bio`
- `struct disk`
- Construction and Destruction
- Optional Methods
- Servicing I/O requests
- Crash dumps

struct bio

- Describes an I/O operation
- `bio_cmd` is operation type
  - `BIO_READ` / `BIO_WRITE`
  - `BIO_FLUSH` – barrier to order operations
  - `BIO_DELETE` – maps to TRIM operations
- `bio_data` and `bio_bcount` describe buffer
- `bio_driver1` and `bio_driver2` are available for driver use

bio Queues

- Helper API to manage pending I/O requests
- `bioq_takefirst()` removes next request and returns it
- `bioq_disksort()` inserts requests in the traditional elevator order
- `bioq_insert_tail()` inserts at tail
- More details in `sys/kern/subr_disk.c`

struct disk

- Various attributes set by driver
  - d_maxsize (maximum I/O size)
  - d_mediasize, d_sectorsize (bytes)
  - d_fwheads, d_fwsectors
  - d_name, d_unit
- Function pointers
- Driver fields
  - d_drv1 (typically softc)

Construction and Destruction

- `disk_alloc()` creates a `struct disk`
- Set attributes, function pointers, and driver fields
- Register disk by calling `disk_create()`, `DISK_VERSION` passed as second argument
- Call `disk_destroy()` to destroy a disk
  - All future I/O requests will fail with `EIO`
  - Driver responsible for failing queued requests

Optional Disk Methods

- `d_open()` is called on first open
- `d_close()` is called on last close
- d_ioctl() can provide driver-specific ioctls
- d_getattr() can provide custom GEOM attributes
  - Return -1 for unknown attribute requests

Servicing I/O Requests

- bio structures passed to `d_strategy()`
- Driver typically adds request to queue and invokes a start routine
- Start routine passes pending requests to the controller
  - Does nothing if using DMA and queue is frozen
- Driver calls `biodone()` to complete request
  - `bio_resid` updated on success
  - `bio_error` and `BIO_ERROR` flag set on failure

Crash Dumps

- Support enabled by providing `d_dump()`
- `d_dump()` is called for each block to write during a crash dump, must use polling
- First argument is a pointer to `struct disk`
- Memory to write described by `_virtual`, `_physical`, and `_length`
- Location on disk described by `_offset` and `_length` (both in bytes)

Example 8: mfi(4)

- `sys/dev/mfi/mfi.c` and `sys/dev/mfi/mfi_disk.c`
- `mfi_disk_attach()` creates a disk
- `mfi_disk_open()` and `mfi_disk_close()`
- `mfi_disk_strategy()`, `mfi_startio()`, and `mfi_disk_complete()` handle I/O requests
- `mfi_disk_dump()`

Conclusion

- Slides and examples available at http://www.FreeBSD.org/~jhb/papers/drivers/
- Mailing list for device driver development is drivers@FreeBSD.org
- Questions?