The Anatomy of a Linux PCIe Network Driver

James Pascoe

james@jamespascoe.com

www.jamespascoe.com

accu
professionalism in programming

Hello and Welcome …

1. Why a talk on PCIe Network Drivers?
   - ‘Anatomy’ in what sense?
   - Why is this useful?
   - Example: drivers/net/wireless/ath/wil6210

2. What are the learning objectives?
   - Disambiguate kernel concepts
   - Be able to read the code for other drivers
   - Implement concepts within your own projects

What we are going to cover …

- PCIe terminology
- Initialisation (complex!)
- Threaded Interrupt Handlers
- The packet path (and sk_buffs)
- Interrupt mitigation with NAPI
- User Agent Interactions (nl80211)
- Driver teardown
- Testing, Optimisation and Advice

Peripheral Component Interconnect Express

- PCIe replaced the PCI-X and PCI standards (high speed serial vs. parallel)
- The PCIe link refers to the connection between one endpoint and another, whereas, a link can exhibit multiple parallel lanes
- PCIe devices have a VID (Vendor ID) and a PID (product ID)

- PCIe endpoints advertise their capabilities within 20ms of power on:
  - Message Signalled Interrupts (MSI): device raises an interrupt by writing a value to a 'special' memory address
  - Memory Mapped IO regions: mapped into the host's address space. On doing so, the host programs the Base Address Registers (BARs) on the device

Initialisation – The Linux Module

```c
static int __init accu_driver_init(void)
{
  int ret;
  ret = pci_register_driver(&accu_driver);
  return ret;
}

static void __exit accu_driver_exit(void)
{
  pci_unregister_driver(&accu_driver);
}

module_init(accu_driver_init);
module_exit(accu_driver_exit);
```

```makefile
# Simple Makefile
obj-m += main.o

KDIR ?= /lib/modules/`uname -r`/build

default:
	$(MAKE) -C $(KDIR) M=$$PWD
```

- Design Makefiles to build the driver code both in-tree and out-of-tree from the start

Initialisation – PCIe Handler Registration

```
#define PCI_VENDOR_ID_ACCU 0xbeef
#define PCI_DEVICE_ID_ACCU 0xcafe

static const struct pci_device_id accu_pcie_ids[] = {
  {PCI_DEVICE(PCI_VENDOR_ID_ACCU, PCI_DEVICE_ID_ACCU)},
  {},  /* Zero entry */
};

MODULE_DEVICE_TABLE(pci, accu_pcie_ids);

static struct pci_driver accu_driver = {
          .probe = accu_pcie_probe,
          .remove = accu_pcie_remove,
          .id_table = accu_pcie_ids,
          .name = “ACCU",
}
```

```
static int accu_pcie_probe(struct pci_dev *pdev, const
                           struct pci_device_id *id)
{
  struct accu *accu = /* kzalloc’d elsewhere */;
  pci_set_drvdata(pdev, accu);
  pci_enable_device(pdev);
  pci_request_region(pdev, ACCU_IO_BAR, “IO");
  accu->io = pci_ioremap_bar(pdev, ACCU_IO_BAR);
  pci_enable_msi(pdev);
  request_threaded_irq(pdev->irq, accu_hardirq,
    accu_threadirq, 0, "accu", accu);
  accu_if_add(accu); /* Initialise buffers */

  return 0;
}
```

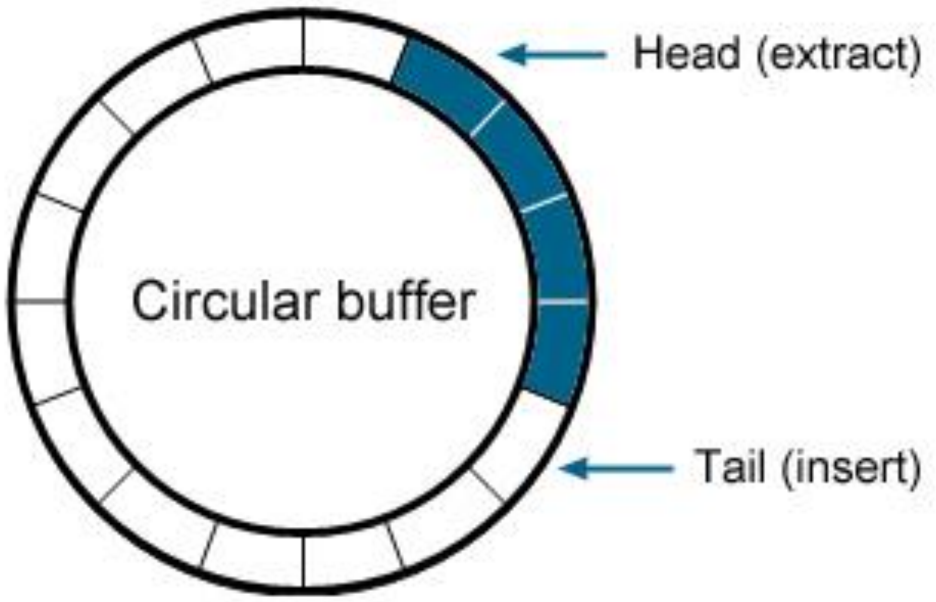
Initialisation – Network Device Registration

```
static int accu_if_add(struct accu *accu)
{
  struct net_device *ndev;
  ndev = alloc_netdev(0, “wlan%d", NET_NAME_UNKNOWN, ether_setup);
  ndev->phy_device = … /* Phy configuration */
  ndev->phy_device->priv = accu;

  ndev->netdev_ops = &accu_netdev_ops;
  netif_napi_add(ndev, &accu->napi_rx,
    accu_netdev_poll_rx, NAPI_POLL_WEIGHT);
  netif_tx_napi_add(ndev, &accu->napi_tx,
    accu_netdev_poll_tx, NAPI_POLL_WEIGHT);

  register_netdev(ndev); /* Viewable with ip link / ifconfig */
  …
}
```

```
static const struct net_device_ops accu_netdev_ops = {
		.ndo_open = accu_open,
		.ndo_stop = accu_stop,
		.ndo_change_mtu = accu_change_mtu,
		.ndo_start_xmit = accu_start_xmit,
		.ndo_set_mac_address = accu_set_mac_address,
		.ndo_validate_addr = eth_validate_addr
};
```

Initialisation – Device Initialisation

```c
static int accu_open(struct net_device *ndev)
{
  struct accu *accu = ndev_to_accu(ndev);

  if (!accu->setup_complete) {
    memset(&accu->rx_ring, 0, sizeof(struct accu_ring));
    memset(&accu->tx_ring, 0, sizeof(struct accu_ring));
    memset(&accu->mgt_ring, 0, sizeof(struct accu_ring));
    accu_allocate_rings(accu);
    accu_send_ring_info_to_hw(accu);
    napi_enable(&accu->napi_rx);
    napi_enable(&accu->napi_tx);
    accu->setup_complete = 1;
  }
}
```



Initialisation – Interrupt Handlers

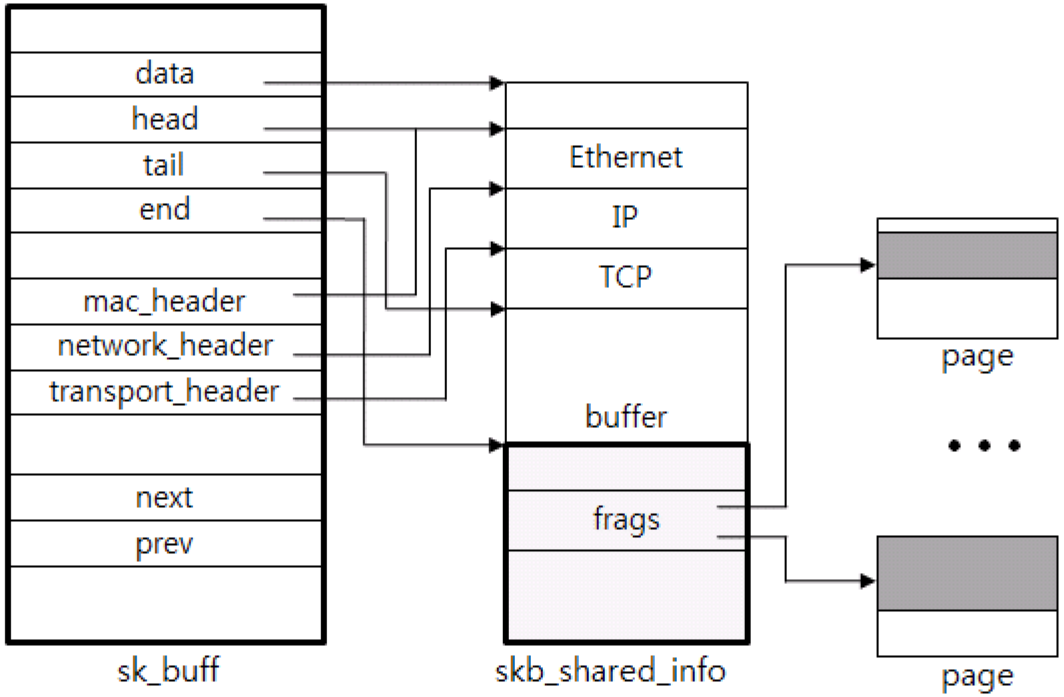
```
static irqreturn_t bh2_hardirq(int irq, void *priv)
{
  irqreturn_t rc = IRQ_HANDLED;
  struct accu *accu = priv;

  /* Check rings */
  if (ring_head(&accu->tx_ring) != ring_tail(&accu->tx_ring))
    napi_schedule(&accu->napi_tx); /* Schedule Tx cleanup */
  if (ring_head(&accu->rx_ring) != ring_tail(&accu->rx_ring))
    napi_schedule(&accu->napi_rx); /* Schedule an Rx event */
  if (ring_head(&accu->mgt_ring) != ring_tail(&accu->mgt_ring))
    rc = IRQ_WAKE_THREAD; /* Dispatch management event */

  return rc;
}
```

```
static irqreturn_t bh2_threadirq(int irq, void *priv)
{
  struct accu_mgt_ring_event *evt = NULL;

  evt = kmalloc(sizeof(struct accu_mgt_ring_event), GFP_ATOMIC);
  evt->entry = ring_head(&accu->mgt_ring);

  mutex_lock(&accu->mgt_evt_mutex);
  list_add_tail(&evt->list, &accu->mgt_pending_events);
  mutex_unlock(&accu->bmi_evt_mutex);
  queue_work(accu->mgt_wq, &accu->mgt_event_worker);

  /* Process all outstanding events in mgt ring */
  return rc;
}
```

And we’ve initialised …

The Packet Path – `struct sk_buff`

- Socket buffer: include/linux/skbuff.h
- Assembling a packet for transmission can involve multiple pieces: data from user-space, headers from the IP stack
- skb_shared_info contains data shared between multiple sk_buffs
- Drivers that support scatter-gather iterate through ‘frags’ and setup a DMA transfer for each



The Packet Path – Transmit

```
static int accu_start_xmit(struct sk_buff *skb,
                           struct net_device *ndev)
{
  struct accu *accu = ndev_to_accu(ndev);
  int nr_frags = skb_shinfo(skb)->nr_frags;
  dma_addr_t skb_phys;
  int frag_len, i;

  if (ring_avail(&accu->tx_ring) < /* some watermark */)
    netif_tx_stop_all_queues(ndev); /* wakes up in Tx cleanup */

  frag_len = skb_headlen(skb);
  skb_phys = dma_map_single(skb->data, frag_len, DMA_TO_DEVICE);
  accu_write_tx_ring_tail(skb_phys, frag_len);

  for (i=0; i<nr_frags; ++i) {
    const struct skb_frag_struct *frag =
      &skb_shinfo(skb)->frags[i];

    frag_len = skb_frag_size(frag);
    skb_phys = skb_frag_dma_map(dev, frag, 0, frag_len,
      DMA_TO_DEVICE);

    accu_write_tx_ring_tail(skb_phys, frag_len);
  }
  /* write end of packet */

  wmb(); /* ensure descriptors are written */

  return NETDEV_TX_OK;
}
```

The Packet Path – Transmit Cleanup

- HW sends packet, receives an acknowledgement and frees device side buffers
- HW advances ring head pointer and triggers MSI interrupt to host

- Hard ISR detects ring head movement and schedules NAPI cleanup
- NAPI handler 'frees' ring entries between head and 'remote head'

- DMA mappings are unmapped using: dma_unmap_page
- skb is freed with: dev_kfree_skb

The Packet Path – Receive

1. During initialisation, the host allocates an sk_buff for each ring entry

2. When a packet arrives:
   a. the device reserves the descriptor at the tail of the Rx ring
   b. the packet is DMA'd into the sk_buff pointed to by the descriptor
   c. the device increments the tail pointer and generates an interrupt to the host

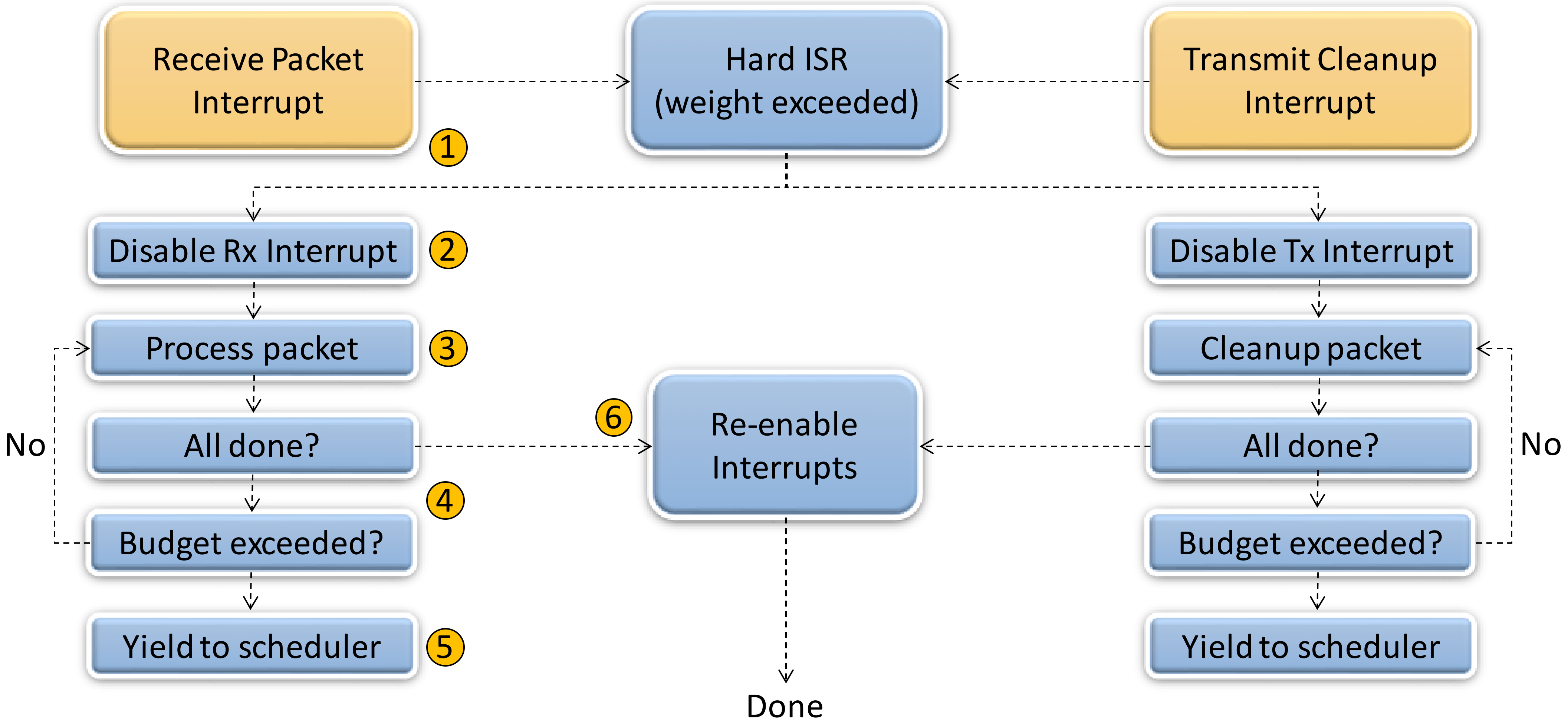
3. When the host receives the interrupt:
   a. the NAPI Rx. Poll handler is scheduled (Rx ring head ≠ tail)
   b. the host inspects the packet's status and updates the sk_buff / netdev statistics
   c. the packet is passed into the network stack with: netif_receive_skb

Interrupt Mitigation – NAPI (New API)

- Simple drivers receive an interrupt for every packet (inefficient)
- Better approach - when the device is busy, disable interrupts and poll
  - Benefit: interrupt load is reduced even though the kernel has to poll
  - Benefit: if the kernel is overrun, then packets are dropped in the device's ring

- To provide fairness, the poll handler is allocated a budget. Once exhausted, it must return control to the NAPI scheduler
- Interrupts are re-enabled when all packets have been processed
- NAPI can also be used to free sk_buffs (Tx cleanup)

Interrupt Mitigation with NAPI

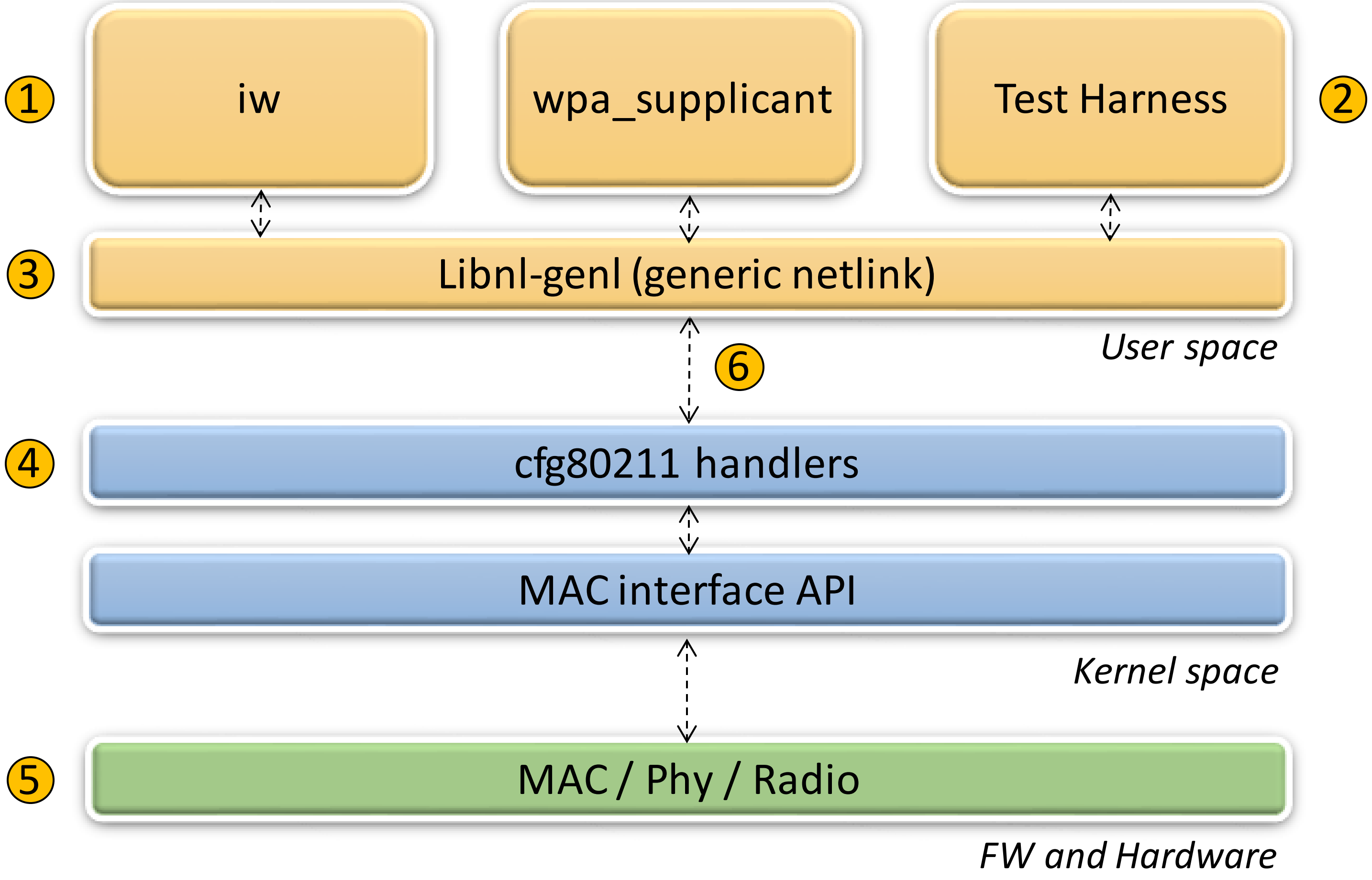


User Agent Communication

- Wireless user agents are programs such as: iw, hostapd, wpa_supplicant
  - Question: how do user agents communicate with the driver?
  - Answer: through nl80211 (user-space) and cfg80211 (kernel)

- nl80211 / cfg80211 replaced Wireless-Extensions
- Based on Netlink sockets (which replaced ioctls)
- Supports lots of wireless operations (e.g. scan, connect etc.)
- Vendor specific commands are also supported
- linux/include/uapi/linux/nl80211.h

The nl80211 Stack



A Note on Teardown

- Two levels: full (module unload) and partial (ifdown)
- The netdev call-backs of interest are:

```
int (*open)(struct net_device *dev);
```

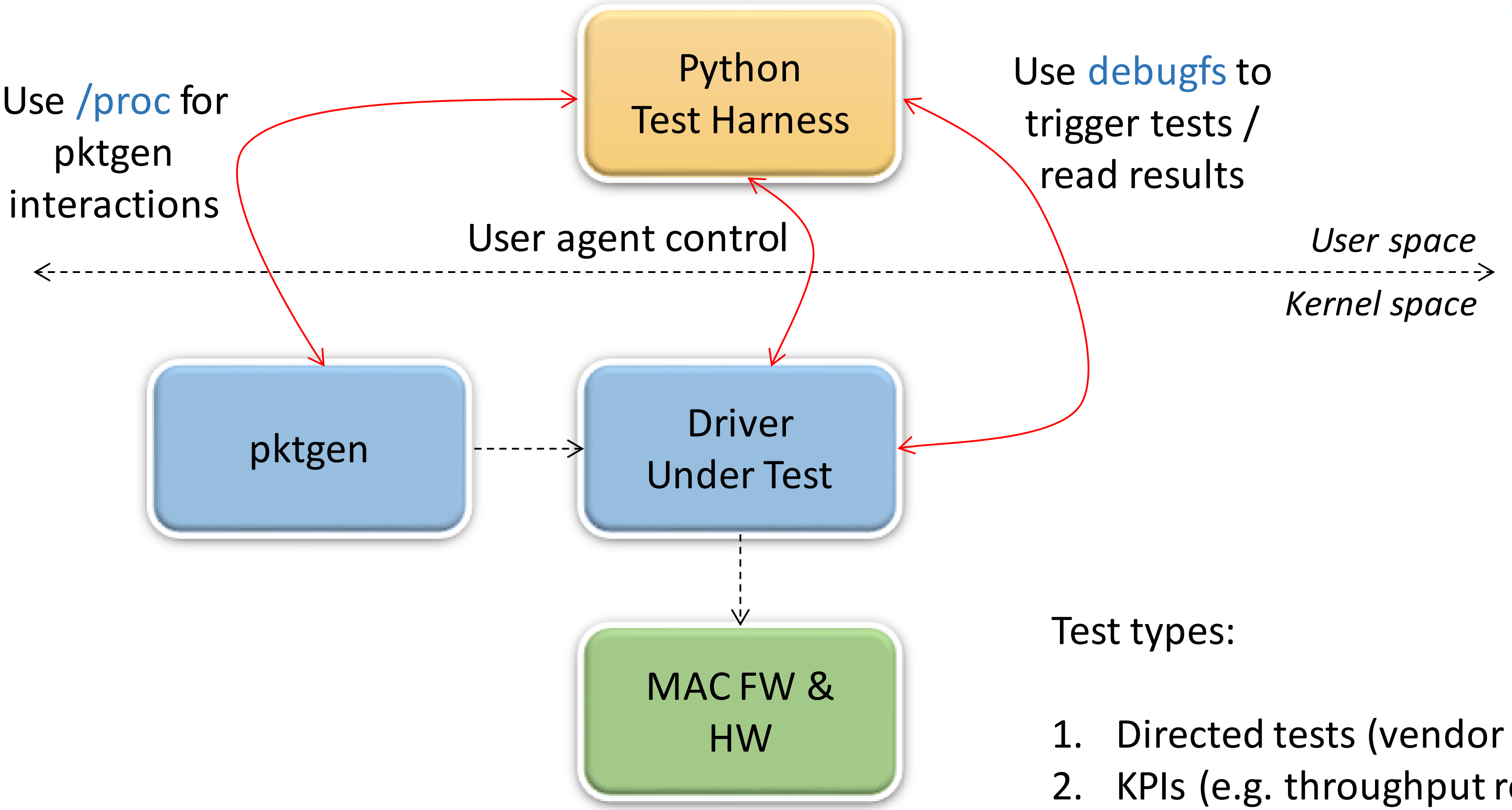
… *the interface is opened whenever ip etc. activates it*

```
int (*close)(struct net_device *dev);
```

… *stops the interface when it is brought down (i.e. reverses open)*

- The close function has to make the hardware quiescent (complex!)
- Keep this is mind as the project progresses

Test Harness



Test types:

1. Directed tests (vendor commands etc.)
2. KPIs (e.g. throughput recipes, latency etc.)
3. Customer specific use-cases
4. Randomised / soak testing

Optimisation

- PCIe latency
  - Minimise PCIe / memory transactions that can not burst
  - PCIe analyzers are invaluable for this
  - Avoid bounce buffers with pci_set_dma_mask

- Interrupt Strategy
  - 1 ISR for all or many interrupts plus more locks?
  - Where possible, defer processing to a worker thread
  - Set NAPI budgets carefully. Rule of thumb is: 64 for ‘fast’ devices, 16 for ‘slow’

- Kernel Internals
  - Export real-time stats over debugfs (and write a nice Python visualizer ☺ !)
  - Use Byte Queue Limits (BQL) to reduce queueing at the NIC
  - Perf + pktgen is a good starting point for network driver optimisation

Advice

- Use a Virtual Machine for development
  - Qemu is brilliant and can be connected to silicon simulators etc.
- Value the git history
  - Use modes such as git add –p to craft beautiful commits
- Implement counters in debugfs
  - Great for debugging the packet path and provides statistics

- Think about thread safety and concurrency from the start
- Use checkpatch.pl –strict for checking commits
- The best documentation really is the Linux kernel source ☺ !

Conclusions

- Writing Linux drivers is an art:
  - … blending kernel abstractions with hardware interactions
- Observations:
  - Linux projects are very collaborative
    - Harness this by ensuring that tools (VMs etc.) and processes (git / checkpatch.pl / test-benches) are available early in the project's life-cycle
  - The best way to learn really is by reading the code
    - Often, the best (only ☺ !) piece of documentation is the header file
    - Invest time in making the code easy to navigate (ctags, vim extensions etc.)
  - Be mindful of deferred aspects (e.g. tear-down) throughout the project

```cpp
#include <iostream>

int main()
{
  std::string questions;
  while (1)
  {
    std::cout << "Questions?" << std::endl;

    if (std::cin >> questions && questions == "Y")
      std::cout << "Answers" << std::endl;
    else
      goto brewdog;
  }

brewdog:
  std::cout << "Thank you for coming !" << std::endl;
}
```