Network device drivers in Linux

Aapo Kalliola
Aalto University School of Science
Otakaari 1
Espoo, Finland
aapo.kalliola@aalto.fi

ABSTRACT
In this paper we analyze the interfaces, functionality and implementation of network device drivers in Linux. We observe that current real-life network drivers are very heavily optimized compared to a minimum baseline implementation. Additionally we analyze some of the optimized functionality in greater detail to gain insight into state-of-the-art network device driver implementations.

Keywords
device driver, network interface, linux

1. INTRODUCTION
Devices in Linux are divided into three main categories: block, character and network devices. Block devices provide aadressable and persistent access to hardware: after writing to a specific block address subsequent reads from the same address return the same content that was written. In contrast, character devices provide unstructured access to hardware. Character device may be a separate devices such as a mouse or a modem or part of the internal computer architecture, for instance a communications bus controller.

Character devices do not typically do buffering while block devices are accessed through a buffer cache. In some cases block devices can also have character device drivers specifically for bypassing the cache for raw I/O.

Network devices in Linux differ significantly from both block and character devices [1]. Data is typically sent by an upper program layer by placing it first in a socket buffer and then initiating the packet transmission with a separate function call. Packet reception may be interrupt-driven, polling-driven or even alternate between these modes depending on configuration and traffic profile. An optimized driver wants to avoid swamping the CPU with interrupts on heavy traffic but the driver also wants to maintain low latency in low traffic scenarios. Therefore it cannot be categorically stated that interrupt-driven would be better than polling or vice versa.

In this paper we will examine network device drivers in Linux starting from driver basics and interfaces and finishing in a partial analysis of certain functionality in the e1000 network driver.

2. OVERVIEW
2.1 Network drivers and interfaces
A network driver is in principle a simple thing: its purpose is to provide the kernel with the means to transmit and receive data packets to and from network using a physical network interface. The driver is loaded into the kernel as a *network module*.

Thus the network device driver code interfaces to two directions: the Linux network stack on one side and device hardware on the other. This is illustrated in Figure 1. While the network stack interface is hardware-independent the interface to network interface hardware is very much dependent on the hardware implementation. The functionality of the network interface is accessed through specific memory addresses, and is thus very much non-generic. For the purposes of this paper we try to remain generic where possible and go into device-dependent discussion only where absolutely necessary.

In Linux kernel source tree network driver code for manufacturers is located in *linux/drivers/net/ethernet/* [2]. Our later analysis of Intel e1000 driver code will be based on the code that is in *linux/drivers/net/ethernet/intel/e1000/*.

2.2 Common data structures and functions
The essential data structures and functions for a network driver.

A network interface is described in Linux as a *struct net_device* item. This structure is defined in <*linux/netdevice.h*>. **TODO:** analyze the most interesting (and later occurring) features of the struct

Packets are handled in transmission as socket buffer structures (*struct sk_buff*) defined in <*linux/skbuff.h*>.

3. DRIVERS IN PRACTISE
In this section we look into network device drivers in the real world. First we look into the minimal functionality a net-

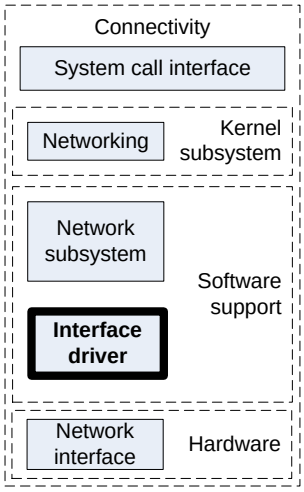

Figure 1: Device driver role in network connectivity

work driver needs to have to transmit and receive packets. Then we look at the e1000 driver for Intel network interface cards and summarize its code organization. Finally we look to into the optimizations and other code that the e1000 driver has in addition to the bare minimum functionality.

3.1 Simple driver

A minimal network device driver needs to be able to transmit and receive packets. In order to be able to do this the driver needs to do a series of tasks: the physical device needs to be detected, enabled and initialized [3]. After initialization the device can be readied for transmitting and receiving packets. For all this to be possible we need to have an understanding of the network device, bus-independent device access, PCI configuration space and the network device chipset's transmission and receive mechanisms.

For device detection it is possible to use the *pci_find_device* function with our device manufacturer's vendor ID and device ID. This gives us a pointer to the device. The device can then be enabled with *pci_enable_device* using the pointer as the only argument.

TODO: some more content about bus-independent memory access and PCI

As previously discussed, network interfaces are seen by the Linux networking stack as devices in *net_device*. This structure is then updated with actual device values during device initialization. In initialization the driver retrieves the base memory address for the device registers, which is needed in all subsequent addressing of device registers.

The transmission and receiving mechanisms on hardware are device-dependent and can be discovered either through manufacturer documentation such as [4] or by reverse engineering. Usual areas of interest are the transmit and receive memory buffers. These buffers can be implemented on hardware as, for instance, ring buffers.

TODO: more details about the receive and transmit control

3.2 e1000

TODO: Continue to do a more thorough and comprehensive overview of the e1000 code and walkthroughs of key functions.

3.2.1 struct e1000_adapter

The structure *struct e1000_adapter* in *e1000.h* is the board-specific private data structure. It contains many interesting fields for features related to VLANs, link speed and duplex management, packet counters, interrupt throttle rate, TX and RX buffers, and also the operating system defined structures *net_device* and *pci_dev*. An adapter may have multiple RX and TX queues active at the same time.

3.2.2 e1000_open

When an e1000 network interface is activated by the system the function *e1000_open* is called. This function allocates resources for transmit and receive operations, registers the interrupt (IRQ) handler with the operating system, starts a watchdog task and notifies the stack that the interface is ready. Since e1000 driver uses NAPI it also calls *napi_enable* before enabling the adapter IRQ.

3.2.3 Buffers

e1000 has receive (RX) and transmit (TX) buffers implemented as ring buffers in *e1000.h*.

The TX buffer *e1000_tx_ring* has a varying size, while the *e1000_rx_ring* also contains a cpu value. **TODO: purpose?**

3.3 Simple driver vs. e1000 driver

TODO: Compare e1000 vs simple driver and try to point out the most glaring/interesting differences.

It is obvious that the e1000 driver is hugely more complex than the simple driver: whereas the simple driver can be implemented in under 1000 lines of code (LOC) the e1000 driver has closer to 20.000 LOC.

e1000	simple	name	description
✓	-	*net_device*	OS network device struct
✓	✓	*pci_dev*	PCI device
-	✓		base I/O address.
✓	✓		TX and RX buffers
✓	-		Device statistics
✓	-	*active_vlans*	Virtual LAN
✓	-	*wol*	Wake-on-LAN
✓	-	*tx_itr, ...*	Interrupt throttling
✓	-		Link speed control
✓	-		HW checksumming
✓	-		NAPI
✓	-		Thread safety
✓	-		Watchdog functions

Table 1: Adapter private structure comparison

One comparison that can be done to get some insight about the feature differences between the simple driver and the e1000 driver is to look into their private data structure and see what fields they contain. These names are shown, if applicable, and grouped into categories with descriptions in Table 1.

4. FEATURES

In this section we look into specific feature implementations in the e1000 driver.

4.1 Interrupt throttling

When the network interface receives a series of packet it can send an interrupt, buffer the packet and wait for a poll later on, or buffer the packet and send an interrupt when the allocated buffer size hits a given threshold. This set of different behaviours is an area that has plenty of opportunities for optimization for different traffic profiles. Polling is typically less resource-consuming with heavy traffic, whereas interrupt-driven operation is better for low latencies when the traffic is not very heavy.

The e1000 driver has an internal interrupt throtte rate (ITR) control mechanism that seeks to dynamically adjust the interrupt frequency depending on the current traffic pattern. This functionality is controlled using the *InterruptThrottleRate* parameter in *e1000_param.c*. Functions relevant to ITR are:

- *e1000_update_itr*
- *e1000_set_itr*
- *e1000_clean* (NAPI related)

It seems that the driver is capable of adjusting not only the interrupt frequency but also whether it works in polling or interrupt-driven mode (*e1000_clean*). **Investigate further**.

TODO: further analyze the state machine and the input parameters

4.2 Protocol offloading

TODO: Checksumming discussion.

TCP Segmentation offloading (TSO) is the functionality that does the the cutting of packets to a maximum of 1500 byte chunks on transmission. The purpose of doing this in the network interface hardware is to lessen the load of the system CPU while maintaining high throughput. Since the operating system must know if the network interface supports TSO the support is defined by the e1000 driver to the hardware feature of the *net_device* structure. **TODO:** continue this..

TCP and UDP checksum offloading is enabled for packet reception in *e1000_configure_rx*. Actual RX checksum calculation is done in *e1000_rx_checksum*. TX checksums are unsurprisingly done in *e1000_tx_csum*. **TODO:** does it generate a tx checksum for UDP??

5. CONCLUSION

Modern network device driver and hardware handles functionality that has originally been the responsibility of the operating system and the CPU, such as packet fragmentation and checksum computation. This leads to significant complexity in the driver compared to a more feature-limited approach.

One of the interesting findings in this paper was that the e1000 driver has significant control over the packet processing mode: it is capable of adjusting its interrupt generation and polling vs. interrupt driven mode selection depending on traffic profile. This effectively places a significant amount of burden on the network interface hardware and driver as weak implementation in them might well hamper the performance of the whole system.

Overall our **still incomplete** analysis of a minimal network device driver and the state-of-the-art e1000 driver our paper shows that the current real-life drivers are thoroughly optimized and feature-rich.

6. REMAINING WORK...

This paper is still very much a work in progress, apologies to the reviewers for this. I'll expand on at least simple driver vs e1000 and features analysis, but in general more details and verification of content currentness (most sources are >5 years old) will be forthcoming.

7. REFERENCES

[1] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. 2005.

[2] http://lxr.linux.no/#linux+v3.6.7/drivers/net/ethernet, November 2012.

[3] M. L. Jangir. Writing network device drivers for linux. *Linux Gazette*, (156), November 2008.

[4] Realtek. *RTL8139(A/B) Programming guide: (V0.1)*. 1999.