# Networking in Biscuit

Sebastian Voit, Paul Emmerich*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: voit@in.tum.de, emmericp@net.in.tum.de

***Abstract*—Biscuit is a High Level Language (HLL) kernel written in Go as a research project to evaluate the impact HLLs have on the performance of operating systems. Go was chosen as it allows easy asm calls, offers good concurrency and can be statically analyzed and compiles to machine code. While Biscuit as a whole has been thoroughly analyzed, this is not true for its components. In this paper we present Biscuit's ixgbe network driver and its implementation. The driver is separated into three main parts: initialization and packet reception and transmission. Afterwards it is compared with ixy.go, an ixgbe user space network driver for Linux systems. Go systems are notably slower (around 15%) then the same systems written in C but offer increased security as memory related bugs cannot occur.**

***Index Terms*—Biscuit, Go, Networking, NIC, Driver**

## 1. Introduction

Today the internet is arguably one of the most important aspects of IT. The ISO/OSI model is well known and describes the process of sending information from one participant to another on a high and abstract level. Network Interface Controller (NICs) are pieces of hardware that implement the necessary functionality to send and receive data using physical and link layer standards and thus provide the base of the ISO/OSI model. NICs are accessed using drivers that allow communication between the physical card and the software running on the computer and therefore are an integral part of Operating System (OS) kernels. These kernels are almost exclusively written in C (or the C family) and assembler, be it a Linux kernel or a Windows NT kernel and thus the drivers are as well. However in recent years there has been an effort to write OS kernels in different languages. Biscuit [1] is such a kernel and is written in the Go programming language and assembler. In this paper we will take a look at its implementation of the ixgbe network driver for the `Intel 82599 10GbE` controller.

Section 2 gives an overview of network drivers in general, more specifically of those written in Go, and the resulting differences. In Section 3 the driver implemented in the Biscuit kernel will be presented. Next we will compare the driver with a user space network driver for the same device family written in Go, ixy.go, in Section 4 and finally, close this paper by pointing out advantages and disadvantages of these drivers in section 5.

## 2. Network Drivers (in Go)

Drivers are pieces of software that allow communication between a hardware device and the rest of the system. The driver abstracts from the actual hardware access and instead exposes interfaces that are better to handle from an outside programmer's perspective. The datasheet for the `Intel 82599` family is freely available online and describes the NIC in its full extent [2].

In case of drivers for network cards there are multiple things that have to be managed:

- Transmit queues (TX) and receive queues (RX) are used by the NIC to receive and send packets. They are organized as ring buffers.
- DMA (Direct Memory Access) memory for the packets which the NIC and the driver can access.
- DMA memory for the packet descriptors. They hold information about the corresponding packet buffers and the packets contained in them and control the behavior of the packet buffers. They are organized as ring buffers.
- Access to the PCI device file in order to control the NIC and enable DMA memory.
- Access to the pagemap (interface for the page tables) as NICs work with physical addresses. It has to be ensured that the mapping stays consistent.

These are general problems for network drivers, independently of whether they are written for the user or kernel space. Specific problems are discussed in Section 4.

Using Go to write network drivers is similar to writing a network driver in C, as the language has intentionally been designed with this similarity in mind. Apart from obvious changes to the syntax it has to be noted that Go does not support many of the tools that are standard in systems programming, specifically in driver programming. These include the volatile operator as reads and writes to registers have to be processed immediately and cannot be cached. Another important aspect of Go is that while it does support pointers, due to its runtime and type safety feature, it does not support pointer arithmetic. In C one usually operates on the allocated DMA memory via pointers. In Go this is only possible via the use of the unsafe package that offers unsafe or arbitrary pointers, which circumvent the runtime as well as many safety measures. Thus care is required in order to break as few assumptions of the runtime as possible when using this approach.


doi: 10.2313/NET-2019-06-1_15

## 3. Architecture

In the following we will take a look at the architecture of biscuit's network driver. We start with a high level description and then go on to further elaborate the details and components. All function names, line numbers, and other code references refer to the file `/biscuit/src/ixgbe/ixgbe.go` of commit `2a2dbe1` of the biscuit github page [1].

### 3.1. Overview

On a high level the driver manages the NIC and its state, transmit and receive queues, handles incoming packets and hands them over to applications on top of the driver, and offers an interface which can be used to send packets.

The driver can be loaded by calling its `Ixgbe_init()` function, which registers the PCI device and its initialization function `attach_ixgbe()` (lines 1272ff.). This function executes the setup procedure: configure flow control, offloading, RX and TX queues and interrupts. After the initialization, received packets will automatically be handed over to the network stack and an interface `Tx_[raw|ipv4|tcp|tcp_tso]()` (lines 911ff.) is provided for sending packets of different types.

### 3.2. Details

After this high level view of the driver's architecture we present a more detailed description of its components. It is recommended to have the implementation and the datasheet at hand. References to sections in the datasheet will be noted as `DS X` where X is the section number.

**3.2.1. Receive and Transmit Descriptors.** The code itself starts with the definition of all relevant constants and corresponding helper functions (lines 1 to 309, DS 8.2 and 8.3). Afterwards the TX and RX descriptors and functions on them are defined. Note that the advanced descriptors are used. Refer to DS 7.1.6 for the Advanced Receive Descriptors and DS 7.2.3.2 for the Advanced Transmit Descriptors. These registers are represented as two 64-bit unsigned integers each, thus reading and writing is done by setting the corresponding bits to 0 or 1. The NIC supports various offloading features such as computing and verifying checksums. The function `ipsumok()` (line 356) operates on the write-back format and checks, in case a non layer two packet was received, whether the bits 6 and 31 of the descriptor's second line are set as this indicates a bad IP checksum.

**3.2.2. Device Properties.** Next, starting from line 615, the status of the device is described and functions are defined that handle and mutate its state. A device has the following properties:

- Pci address (tag):
  The pci address the device is located at. Most importantly the Base Address Registers (BARs) addresses are exposed via pci.
- BAR0 address (bar0):
  The BARs expose configuration and control registers to the drivers. While the NIC's address space is mapped into multiple memory regions, only the BAR0 is necessary as described in DS 8.1. This address space is mapped in `init()` (line 650ff.) and accessible via the bar0 slice.
  The functions `rs()` and `rl()` (lines 681ff. and 688ff.) write to and read from the registers as an offset of bar0.
- Transmit queues (txs):
  Queues that are used for packet transmissions. Contains TX descriptors and their current number. The queue tail as well as some additional parameters are cached to reduce expensive register reads.
  Packets can be enqueued for transmission with the `Tx_[raw|ipv4|tcp|tcp_tso]()` (lines 911ff.) functions. Note that sending is asynchronous: an enqueued packet does not have to be sent out immediately but the NIC will set the DD flag of the TX descriptor once it has been sent out.
- Receive queue (rx):
  Queue that are used for packet receptions. Contains RX descriptors, their current number, a slice referencing the packets and the queue tail is cached.
- Number of allocated pages (pgs):
  Incremented by one whenever a page is added (see `pg_new()`, line 89ff.).
- Link status (linkup):
  Whether the NIC is operational.

This is not the full list but includes most that are relevant for a general understanding. Refer to the implementation for the full list.

**3.2.3. Sending.** Next we will take a more detailed look at sending. The `Tx_[raw|ipv4|tcp|tcp_tso]()` functions (note that these are the exported functions as they start with a capital letter) all call `_tx_nowait()` (lines 927ff.) with the corresponding arguments. This function locks a TX queue and calls `_tx_enqueue()` (lines 963ff.) which handles the actual sending. It takes information about the packet to send and tries to enqueue the packet in the transmission queue. The function returns true on success.

First the packet buffer is checked for empty rows which will be deleted and parameters are checked for correctness. The hardware controls the head pointer and the driver the tail pointer of the ring buffer. The next step is to find out how many buffers are needed and whether there are enough that are free for use. The DD (descriptor done) flag of the status register is set when the descriptor is done, indicating that the packet has been sent out and the descriptor can be reused. The eop (end of packet) flag is set if it is the last descriptor of a packet (see DS 7.2.3.2.4 for the flags). If there are not enough descriptors for the packet buffer, the function returns false, else there is enough space in the transmit queue and the packet can be enqueued.

For sending, the packet headers will be handled first and the rest of the packet afterwards, refer to lines 1046 ff. for the enqueueing. Depending on the type of packet (ethernet, ipv4, etc.) the approach has to be different to accommodate to the packet properties. After the header is done, the payload can be treated independently. The

1. https://github.com/mit-pdos/biscuit/tree/2a2dbe1228881c94764f1cdf6134dca27defab12


doi: 10.2313/NET-2019-06-1_15

only thing left is to update the tail pointer so that the NIC can now send out the packets in the queue.

**3.2.4. Receiving.** Now that we have discussed how sending packets works, the next step is receiving packets. This is implemented in `rx_consume()` in lines 1086ff. Receiving works on a RX queue which is organized similar to a TX queue. A memory area that is handled as a ring buffer with a head pointer controlled by the hardware and a tail pointer controlled by the driver. When processing received packets it first has to be checked for the DD flag, which indicates a received packet, until the current head is found. In the case of no newly received packets the function is done. Note that the tail itself is empty and thus has to be skipped. For each descriptor DMA memory in the size of the packet is allocated and the packet is then handed over to the network stack. Afterwards the descriptors are reset and the tail pointer update is sent to the NIC.

## 3.3. Interrupt Handling

When operating a NIC interrupts may be triggered which need to be handled by the driver. For this driver interrupt handling is implemented in int_handler() (lines 1151ff.). Interrupts are described in DS 7.3. First the interrupt handle has to be registered. The code then runs in a forever loop: wait for an interrupt and handle it.

Four different types of interrupts in the Extended Interrupt Cause Register (EICR, DS 7.3.1.1 for the description and DS 8.2.3.5.1 for the register) are handled. Note that the queue interrupts are mapped to bit 0 for all RX queues and to bit 1 for all TX queues (lines 1406-1409, DS 8.2.3.5.16).

- RX queue interrupt (bit 0):
  Is raised on descriptor write back, a full queue or upon reaching a minimum threshold. Thus as this indicates newly received packets, they are to be retrieved via a call to `rx_consume()`
- TX queue interrupt (bit 1):
  Is raised on descriptor write back. This indicates that packets have been sent out but as sending is done via the corresponding functions, nothing has to be done. It has to be assumed that this is left over from the programming process.
- Rx Miss (bit 17):
  Is raised when packets are dropped due to a full Rx queue (overrun). Nothing additional that can be done as rx_consume() would already be running and packets arrive faster than they are being sent out; increment statistic.
- Link Status Change (bit 20):
  Is raised when the link status changes, e.g. from down to up or vice versa. The new status is printed, the NIC is tested and a goroutine is started that periodically prints the number of received and dropped packets.

**3.3.1. Setting Up the NIC.** Now that the operations of the driver on the NIC are defined, the only thing that is left is the setup of the device. This is done in the `attach_ixgbe()` function (lines 1272ff.). DS 4.6.3 describes this procedure. Please refer to the datasheet for flag and register names and other details which we cannot cover here. Note that the steps in the driver are not necessarily is the same order as proposed in the datasheet as reordering can be more efficient, as long as it does not influence the result e.g. PHY is reset before waiting for the DMA initialization as the latter has no influence on the former.

1) Disable Interrupts and call `init()` (lines 650ff.) on a new `ixgbe_t` struct which from then on represents the device. After the reset disable interrupts again (DS 4.6.3.1).
2) As flow control is disabled, the registers FCTTV, FCRTL, FCRTH, FCRTV and FCCFG are set to 0x0 (DS 4.6.3.2) and the assumption of disabled flow control is checked.
3) No snoop is enabled. Processor caches do not have to be snooped in this case and direct access to the DRAM is faster.
4) The physical address is reset via MDI command: the MSCA register (8.2.3.22.11) allows the use of the MDIO interface (3.7.6) with which physical registers can be accessed. As clause 45 operations are utilized, op code `00b` has to be sent first and afterwards `11b` for the read (DS 3.7.6.4).
5) Wait for the DMAIDONE flag of the RDRXCTL register.
6) Load the MAC address from the Receive Address Registers and cache it.
7) Enable Message Signaled Interrupts (MSI) via PCI and ensure that legacy interrupts are disabled. Reset all interrupts (EIAC register), disable automask (EIAM registers), disable interrupt throttling (EITR(n)), and map all RX queues to EICR bit 0 and all TX queues to EICR bit 1 (4.6.6).
8) Disable Receive Side Coalescing (RSC), a technique that would accumulate TCP/IP packets that belong to the same flow into large packets [3].
9) Enable and configure receive queues (4.6.7):
   a) Disable VLAN features PFVFSPOOF, MPSAR, PFUTA, PFVLVFB (4.6.10) and VFTA (7.4.4).
   b) Enable ethernet boardcast packets for ARP functionality (FCTRL bit 10).
   c) Offloading: IP checksum (RXCSUM bit 12, 8.2.3.7.5), strip CRC (RDRXCTL bit 1) and bit 12 of the DCA control register must be set to 0.
   d) Setup RX queue:
      i) Allocate new page for queue and send address and size to NIC.
      ii) Calculate number of descriptors and allocate a new packet buffer for each (2048B buffers, two per page).
      iii) Disable header splitting (SRRCTL bits 25:27), write descriptors to the NIC and initialize the receive head pointer (RDH).
      iv) Enable the queue (RDRXCTL bit 25), set the receive tail pointer (RDT) and enable receive (RXCTRL bit 0).
10) Enable and configure transmit queues (4.6.8):
   a) Map all TX queue statistics to a single counter.
   b) Enable layer two offloading via HLREG0 (7.1.3): CRC offloading (bit 0) and stripping (bit 1), padding (bit 10) and receive length errors (bit


doi: 10.2313/NET-2019-06-1_15

27).

c) `_dbc_init()` (actually referring to DCB, Data Center Bridging, see 4.6.11 for the configuration and 7.7 for the description) (lines 1779ff.):
   - Implements DCB-off, VT-off (4.6.11.3.4) as neither flow control not virtualization is supported.

d) Setup multiple TX queues (default four):
   i) Number each queue, allocate a new page for the descriptors and send address and size to NIC.
   ii) Calculate number of descriptors and allocate a new packet buffer for each (2048B buffers, two per page) and set the DD and eop flags so they are ready for use.
   iii) Disable head write-back of the queue.
   iv) Transmit Control (TXDCTL(queue_id)): Set thresholds for Pre-Fetch, Host and Write-Back. These values orient themselves at the number of descriptors that fin in a cache line to avoid cache thrashing.
   v) Initialize descriptor head and tail.

e) Enable transmission (DMATXCTL bit 0).

f) Enable transmission queues (TXDCTL(queue_id) bit 25) and wait for success.

11) Configure and enable interrupts:

a) Set the General Purpose Interrupt Enable register (GPIE, 8.2.3.5.18) to 0. This, among others, configures the use of MSI interrupts and clears the EICR register on read and disables many unneeded features.

b) Set the interrupt throttle to a 125$\mu$s as lower values can have significant impact on performance, especially within TCP bulk transfer.

c) Clear previous interrupts (EICR) and start the interrupt handler as a goroutine.

d) Enable transmit and receive queue interrupts as well as link change interrupts while disabling all other types of interrupts (EIMS).

The rest of the code are testing functions which we will not discuss in this work.

## 4. Comparison with ixy.go

Another driver for `Intel 82599 10 GbE` Controller written in Go is ixy.go [4]. This is a user space driver for Linux operating systems. This means that it runs completely in user space compared to Biscuit's ixgbe driver that is part of the kernel. Therefore while the NIC is still programmed in the same way, the approach differs at times. We will not consider differences in the functionality of the drivers as ixy.go is meant to be an educational driver and thus is intentionally kept simple and without much of the functionality that a driver for a running kernel needs. From a high point of view, there is not much difference to be found: On startup the NIC is initialized by programming the registers. Afterwards received packets are handled and packets can be send via an interface. Table 1 lists high level stats of both systems. However writing a kernel driver versus writing a user space driver imposes two mayor differences:

1) A user space driver does not have access to privileged system functions and must use syscalls instead.
2) While Biscuit's driver has to provide general purpose packet processing by itself for the rest of the system, ixy.go offers an API with explicit memory allocation, batching and abstraction that is similar to DPDK [5]

Two main challenges arise from the first point. As new pages cannot simply be allocated from the user space, this has to be done via Mmap() from the syscall package [6]. This also changes the way memory is administered. The second challenge is the page virtualization. In Biscuit's driver a new physical page is allocated but from the user space only virtual pages can be allocated. The mapping virtual to physical addresses for the NIC via the pagemap is not an issue but the page migration algorithm can change this mapping at any time. Fortunately, it is not implemented for huge pages which are thus used to keep the physical addresses consistent.

The second difference is mainly an architectural one. Biscuit's network driver automatically checks for incoming packets upon interrupts and hands them over to the network stack. ixy.go on the other hand offers the RxBatch() function which checks for received packets and hands them back in the provided buffer. It is the responsibility of applications built on top of ixy.go to regularly check for incoming packets instead of the driver. Sending is more similar with the main difference being batching. ixy.go has a virtual copy of the packet ring as reading flags is a rather expensive operation. When sending a batch of packets, it is first checked whether a batch of packets has been sent out, reducing the register access to once per batch and afterwards enqueueing as many packets as possible. Biscuit's driver instead checks the descriptor once per previously sent packet (eop is cached) until enough space is found but does so for each packet.

## 5. Conclusion

In this paper we took a look at the network driver of the biscuit kernel. Biscuit has been developed as a scientific operating system to test the impact of higher level languages on operating system kernels. While the system runs notably slower due to the garbage collection and runtime, this impact might still be acceptable for certain use cases. On the other hand Go also brings a distinct set of advantages such as memory safety. Advantages as well as disadvantages also extend to the network driver. Here speed is key and Go is slower then C. The user space driver ixy.go [4] showcases a clear loss in performance compared to its parent project written in C (10-20% depending on batch size and CPU speed). Cutler et al. also found that the kernel as a whole suffered a performance loss of 15% [1] compared to a C kernel. Still, speed is not everything, a driver also has to be secure. Cutler at al. analyzed 65 code execution vulnerabilities in the Linux kernel, 40 of which would have been prevented when using Go. In the end the choice of language always results in a trade-off between speed and security.

Because of these trade-offs we argue that is important to re-implement existing software in other programing languages. In this case Go offers security mechanisms where C programs are vulnerable. Re-implementations can be evaluated and their advantages and disadvantages


doi: 10.2313/NET-2019-06-1_15

| | biscuit | ixy.go |
|---|---|---|
| lines | 1900 (1600 without register offsets) | 1000 |
| unsafe pointer | rx & tx descriptor structs | register access, physical address calculation |
| memory allocation | physical pages | mmap(2) syscall |
| application area | provide packet receive & send functionality to kernel | low level API for fast packet processing |

TABLE 1: Comparison of Biscuit's ixgbe driver and ixy.go

quantified. Similar to the CAP theorem [7] there are always properties that are more important then others depending on the system. Thus having the same programs with different properties lead to generally better systems.

## References


[1] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a POSIX kernel in a high-level language," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, (Carlsbad, CA), pp. 89–105, USENIX Association, 2018.

[2] Intel, "Intel 82599 10 gbe controller datasheet rev. 3.3." https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf. Last visited 30.11.2018.

[3] S. Makineni, R. Iyer, P. Sarangam, D. Newell, L. Zhao, R. Illikkal, and J. Moses, "Receive side coalescing for accelerating tcp/ip processing," in *High Performance Computing - HiPC 2006* (Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, eds.), (Berlin, Heidelberg), pp. 289–300, Springer Berlin Heidelberg, 2006.

[4] S. Voit and P. Emmerich, "Writing network drivers in go," 2018.

[5] DPDK Project, "Dpdk website." https://dpdk.org/. Last visited 14.12.2018.

[6] Go Project, "Go syscall package." https://golang.org/pkg/syscall/. Last visited 14.12.2018.

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent available partition-tolerant web services," in *In ACM SIGACT News*, p. 2002, 2002.