УДК 004.454

РАЗРАБОТКА ДРАЙВЕРА СЕТЕВОЙ КАРТЫ INTEL E1000 ДЛЯ ЛЮБИТЕЛЬСКОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ

Голубчиков И.А., студент гр. ИТб-221, IV курс
Научный руководитель Алексеева Г.А., старший преподаватель
Кузбасский государственный технический университет имени Т.Ф. Горбачёва
г. Кемерово

В современном мире сетевое взаимодействие стало не просто обыденностью, а базово необходимым функционалом для многих вычислительных устройств, в частности для персональных компьютеров. Данный факт стал причиной, по которой в ходе разработки любительской операционной системы «SomniaOS» было принято решение о реализации базового сетевого драйвера.

В качестве целевого сетевого контроллера был выбран Intel e1000. В данной статье под этим термином подразумевается программная модель сетевого адаптера, соответствующая спецификациям Intel PCI/PCIe Network Controllers (семейство 8254x).

Выбор данного типа контроллера обоснован относительной простотой его архитектуры и наличием как официального руководства для разработчиков (Intel Software Developer's Manual), так и большого количества проектов с открытым исходным кодом, которые могут выступать в роли эталона. Помимо этого, архитектура e1000 является фактическим стандартом для большинства современных гигабитных карт.

Основным критерием успешной реализации определим возможность выполнения функционала утилиты Ping – отправку ICMP-запроса (echo request) к целевому хосту и получение от него ICMP-ответа (echo reply).

Рассмотрим архитектуру контроллера.

Контроллер подключается через PCI/PCIe шину и представляется в системе как стандартное PCI-устройство. Обнаружение нужного устройства на шине происходит через уникальную пару параметров Vendor ID и Device ID. В нашем случае эти параметры будут иметь значения 0x8086 и 0x100e соответственно. После обнаружения устройства, необходимо произвести считывание базовых адресных регистров (Base Address Register, BAR), один из которых содержит адрес области регистров устройства.

Управление устройством происходит через обычные операции чтения и записи памяти. Это возможно благодаря механизму MMIO (memory-mapped I/O) при

котором область регистров устройства отображается в адресное пространство процессора.

Среди всех управляющих регистров, можно выделить:

1) Регистры приема:
   - RDBAL / RDBAH – младшая и старшая части адреса массива RX-дескрипторов
   - RDLEN – размер кольца RX-дескрипторов
   - RDH – текущая позиция обработки (head)
   - RDT – позиция хвоста кольца (tail)
   - RCTL – регистр управления приёмом
2) Регистры передачи:
   - TDBAL / TDBAH – младшая и старшая части адреса массива TX-дескрипторов
   - TDLEN – размер кольца TX- дескрипторов
   - TDH – текущая позиция обработки (head)
   - TDT – позиция хвоста кольца (tail)
   - TCTL – регистр управления приёмом
3) Регистры MAC-адреса:
   - RAL / RAH – младшая и старшая части MAC-адреса

Прием и передача сетевых пакетов в контроллере реализованы через механизм дескрипторных колец. Драйвер выделяет в памяти массив дескрипторов фиксированного размера, каждый из которых содержит указатель на буфер данных. Эти массивы образуют кольцевые буферы для передачи (TX) и приёма (RX).

Работа кольца основана на двух указателях – head (позиция, которую обрабатывает контроллер) и tail (позиция, обновляемая драйвером). Драйвер помещает в кольцо дескрипторы с подготовленными буферами и перемещает указатель tail. Контроллер последовательно обрабатывает дескрипторы, выполняя передачу или приём данных, и обновляет указатель head.

На рисунке 1 представлены структуры дескрипторов приема и передачи данных, а на рисунке 2 – структуры дескрипторных колец. Изображения взяты из Intel PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual.

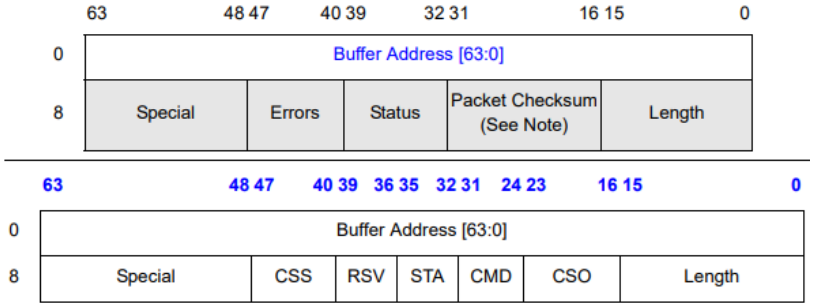


Рисунок 1 – Структуры дескрипторов приема (сверху) и передачи (снизу) данных

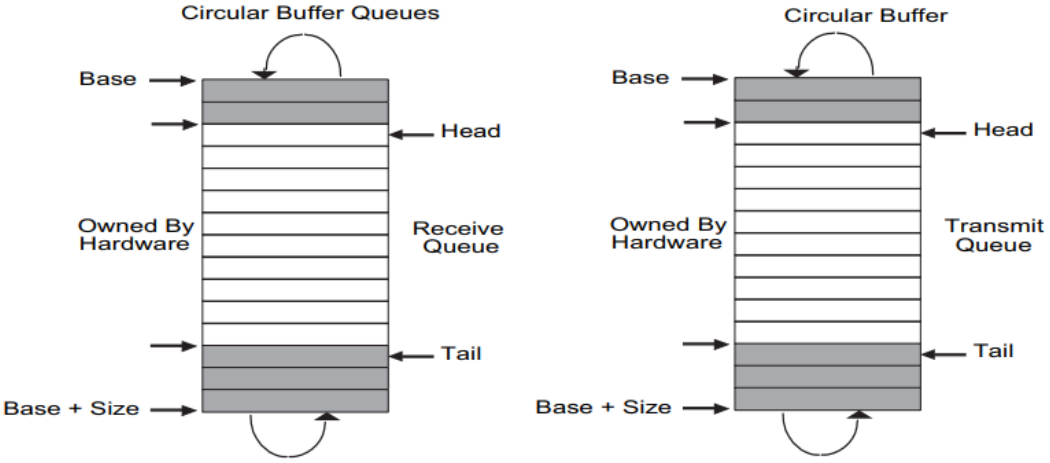


Рисунок 2 – Структуры дескрипторных колец приема (слева) и передачи (справа) данных

Описание значений полей дескрипторов:

1) Общие:
   - Buffer Address – физический адрес буфера в памяти, куда контроллер записывает данные принятого/отправляемого пакета
   - Length – длина принятого/отправляемого пакета в байтах
   - STA / Status – флаги состояния
   - Special – дополнительные данные (например, VLAN-тег или приоритет)
2) Дескриптор передачи:
   - CSO – смещение в пакете, с которого контроллер должен вычислить контрольную сумму
   - CMD – битовая маска команд управления передачей
   - RSV – зарезервированное поле, должно иметь значение 0
   - CSS – смещение, с которого начинаются данные для вычисления контрольной суммы
3) Дескриптор приема:
   - Checksum – вычисленная контроллером контрольная сумма для проверки целостности пакета
   - Errors – битовая маска ошибок, возникших при приеме пакета

Ознакомившись с архитектурой контроллера, можно приступить к реализации драйвера устройства. В качестве языка разработки будем использовать Rust, поскольку именно на нем ведется разработка SomniaOS.

Реализуем структуры данных, необходимые для работы драйвера: дескрипторы приема и передачи, кольцевые буферы и структуру контроллера e1000. Кроме того, для корректной работы драйвера необходимо описать структуры ряда сетевых протоколов: заголовок IPv4 и пакет ARP (на основе Intel Software Developer's Manual), сообщение ICMP Echo/Echo Reply (на основе RFC 792 от сентября

1981) и заголовок Ethernet-кадра (на основе стандарта IEEE 802.3). Реализация указанных структур приведена на рисунке 3.

```rust
const RX_RING: usize = 32;
const TX_RING: usize = 8;
const RX_BUFFER_SIZE: usize = 2048;
static mut RX_DESC: [RxDesc; RX_RING] = [RxDesc { addr:0, length:0, checksum:0, status:0, errors:0, special:0 }; RX_RING];
static mut TX_DESC: [TxDesc; TX_RING] = [TxDesc { addr:0, length:0, cso:0, cmd:0, status:1, css:0, special:0 }; TX_RING];
static mut RX_BUF: [[u8; RX_BUFFER_SIZE]; RX_RING] = [[0; RX_BUFFER_SIZE]; RX_RING];
static mut TX_BUF: [[u8; RX_BUFFER_SIZE]; TX_RING] = [[0; RX_BUFFER_SIZE]; TX_RING];

#[repr(C, align(16))]
#[derive(Copy, Clone)]
pub struct RxDesc {
    pub addr: u64,
    pub length: u16,
    pub checksum: u16,
    pub status: u8,
    pub errors: u8,
    pub special: u16,
}

#[repr(C, align(16))]
#[derive(Copy, Clone)]
pub struct TxDesc {
    pub addr: u64,
    pub length: u16,
    pub cso: u8,
    pub cmd: u8,
    pub status: u8,
    pub css: u8,
    pub special: u16,
}

#[repr(C, packed)]
#[derive(Clone, Copy)]
pub struct IcmpEcho {
    pub icmp_type: u8,
    pub code: u8,
    pub checksum: u16,
    pub ident: u16,
    pub seq: u16,
}

#[repr(C, packed)]
pub struct EthernetHeader {
    pub dst: [u8;6],
    pub src: [u8;6],
    pub ethertype: u16,
}

pub struct E1000 {
    regs: *mut u32,
    rx_desc: &'static mut [RxDesc; RX_RING],
    tx_desc: &'static mut [TxDesc; TX_RING],
    rx_buf: &'static mut [[u8; RX_BUFFER_SIZE]; RX_RING],
    tx_buf: &'static mut [[u8; RX_BUFFER_SIZE]; TX_RING],
    rx_tail: usize,
    tx_tail: usize,
    mac: [u8;6],
    ip: [u8;4],
}

#[repr(C, packed)]
#[derive(Clone, Copy)]
pub struct Ipv4Header {
    vihl: u8,
    tos: u8,
    len: u16,
    id: u16,
    frag: u16,
    ttl: u8,
    proto: u8,
    checksum: u16,
    src: [u8;4],
    dst: [u8;4],
}

#[repr(C, packed)]
#[derive(Clone, Copy)]
pub struct ArpPacket {
    htype: u16,
    ptype: u16,
    hlen: u8,
    plen: u8,
    oper: u16,
    sha: [u8;6],
    spa: [u8;4],
    tha: [u8;6],
    tpa: [u8;4],
}
```

Рисунок 3 – Реализация структур данных, необходимых для работы драйвера

Для структуры E1000 определим следующий набор функций:

- read / write – базовые методы чтения и записи данных для заданных управляющих регистров.
- read_mac – функция чтения MAC-адреса устройства. Извлекает из регистров RAL (первые 32 бита) и RAH (старшие 16 бит) аппаратный адрес, формируя массив из 6 октетов в порядке от младшего к старшему, соответствующем стандартному представлению MAC-адреса.
- init – основная функция инициализации устройства на основе MMIO-области памяти и IP-адреса. Определяет MAC-адрес устройства и настраивает кольцевые буферы.

- init_from_pci – функция-обертка, которая перед вызовом init определяет нужное устройство на PCI шине, активирует его и находит базовый адресный регистр для передачи в параметре.
- init_rx – настройка кольцевого буфера приема. Для каждого дескриптора в кольце устанавливается адрес соответствующего буфера данных в памяти, а поле статуса обнуляется. Затем физический адрес начала кольца дескрипторов записывается в регистры RDBAL/RDBAH, размер кольца – в RDLEN, указатели head (RDH) и tail (RDT) инициализируются нулем и значением RX_RING-1 соответственно. Завершается метод настройкой регистра управления приемом RCTL: включается прием пакетов (бит 1) и разрешается прием широковещательных сообщений (бит 15).
- init_tx – настройка кольцевого буфера передачи, аналогичная инициализации RX. Дескрипторы помечаются как свободные (status = 1), адрес кольца и его размер записываются в соответствующие регистры, указатели head и tail обнуляются. В регистр TIPG заносится значение межпакетного интервала (0x0060200A). Регистр управления TCTL настраивается на включение передачи, дополнение коротких пакетов до минимального размера и задание параметров коллизий.
- send – функция отправки Ethernet-кадра. Получает индекс текущего элемента в TX-кольце, копирует данные во временный буфер и устанавливает адрес буфера в дескрипторе. При необходимости, пакет дополняется нулями до размера в 60 байт, чтобы соответствовать минимально допустимой длине. В дескрипторе указываются cmd-флаги: EOP (End of Packet) – признак последнего дескриптора для пакета, IFCS (Insert FCS) – требование аппаратно добавить контрольную сумму кадра, RS (Report Status) – запрос на обновление поля статуса после отправки. Поле статуса обнуляется, индекс tail увеличивается, и новое значение записывается в регистр TDT, сигнализируя контроллеру о готовности пакета к отправке.
- recv – функция приема Ethernet-кадра. Проверяет текущий дескриптор в RX-кольце: если младший бит поля status установлен в значение 1, значит контроллер записал в буфер принятый пакет. Функция извлекает длину пакета из поля length дескриптора и возвращает ссылку на данные в буфере. После обработки пакета статус дескриптора сбрасывается, индекс tail обновляется, а в регистр RDT записывается номер обработанного дескриптора – это возвращает дескриптор во владение контроллера для приема новых пакетов.

Реализация некоторых описанных функций представлена на рисунке 4.

```rust
pub fn send(&mut self, data: &[u8]) {
    let i = self.tx_tail;
    self.tx_buf[i][..data.len()].copy_from_slice(data);
    self.tx_desc[i].addr = self.tx_buf[i].as_ptr() as u64;

    let len = core::cmp::max(60, data.len());
    self.tx_buf[i][..data.len()].copy_from_slice(data);
    for b in &mut self.tx_buf[i][data.len()..len] {
        *b = 0;
    }

    self.tx_desc[i].length = len as u16;
    self.tx_desc[i].cmd =
        (1 << 0) | // EOP
        (1 << 1) | // IFCS
        (1 << 3);  // RS

    self.tx_desc[i].status = 0;
    self.tx_tail = (self.tx_tail + 1) % TX_RING;
    self.write(REG_TDT, self.tx_tail as u32);
}

pub fn init(regs: *mut u32, ip: [u8;4]) -> Self {
    let mut dev = Self {
        regs,
        rx_desc: unsafe { &mut RX_DESC },
        tx_desc: unsafe { &mut TX_DESC },
        rx_buf: unsafe { &mut RX_BUF },
        tx_buf: unsafe { &mut TX_BUF },
        rx_tail: 0,
        tx_tail: 0,
        mac: [0;6],
        ip,
    };

    // reset
    dev.write(REG_CTRL, 1 << 26);

    for _ in 0..100000 {
        core::hint::spin_loop();
    }

    dev.mac = dev.read_mac();
    dev.init_rx();
    dev.init_tx();
    dev
}
```

```rust
pub fn recv(&mut self) -> Option<&[u8]> {
    let i = self.rx_tail;
    if self.rx_desc[i].status & 1 == 0 {
        return None;
    }

    let len = self.rx_desc[i].length as usize;
    let packet = &self.rx_buf[i][..len];
    self.rx_desc[i].status = 0;
    self.rx_tail = (self.rx_tail + 1) % RX_RING;
    self.write(REG_RDT, i as u32);
    Some(packet)
}

pub fn init_from_pci(ip: [u8;4]) -> Self {
    let dev = pci::find_device(0x8086, 0x100e).unwrap();
    dev.enable();
    let bar0 = dev.bar0();
    let mmio = bar0 as *mut u32;
    E1000::init(mmio, ip)
}

fn init_rx(&mut self) {
    for i in 0..RX_RING {
        self.rx_desc[i].addr = self.rx_buf[i].as_ptr() as u64;
        self.rx_desc[i].status = 0;
    }

    let addr = self.rx_desc.as_ptr() as u64;
    self.write(REG_RDBAL, addr as u32);
    self.write(REG_RDBAH, (addr >> 32) as u32);
    self.write(REG_RDLEN, (RX_RING * core::mem::size_of::<RxDesc>()) as u32);
    self.write(REG_RDH, 0);
    self.write(REG_RDT, (RX_RING - 1) as u32);

    let rctl =
        (1 << 1) | // enable
        (1 << 15); // broadcast accept

    self.write(REG_RCTL, rctl);
}
```

Рисунок 4 – Реализация некоторых методов структуры E1000

Помимо рассмотренных выше механизмов инициализации устройства и работы с дескрипторными кольцами, реализация драйвера включает набор вспомогательных функций, обеспечивающих формирование и обработку сетевых пакетов. Подробный разбор этих функций не может быть выполнен в рамках данной статьи в виду ограничений по размеру, однако кратко опишем их назначение.

Часть вспомогательного кода отвечает за построение сетевых пакетов различных уровней. В частности, реализованы процедуры формирования Ethernet-кадра, заголовка IPv4 и сообщений протокола ICMP. Для расчёта контрольных

сумм используется отдельная функция, реализующая стандартный алгоритм 16-битной контрольной суммы, применяемый в протоколах IPv4 и ICMP.

Другой важной частью является реализация минимальной поддержки протокола ARP, необходимой для определения MAC-адреса узла по его IP-адресу. Для этого драйвер формирует ARP-запрос и анализирует входящие пакеты до получения соответствующего ARP-ответа. Полученный MAC-адрес затем используется при формировании Ethernet-кадров более высокого уровня.

Также в коде присутствует логика обработки входящих пакетов. После приёма кадра выполняется его первичный разбор: определяется тип протокола и выполняется дальнейшая обработка на уровне IP и ICMP. Такой механизм позволяет, в частности, корректно отвечать на ICMP Echo Request, что делает возможным тестирование работы драйвера с помощью утилиты ping. Демонстрация такого тестирования приведена на рисунке 5. Пакеты успешно проходят от одной виртуальной машины под управлением SomniaOS к другой, что видно, в том числе в программе анализа сетевого трафика Wireshark на рисунке 6.

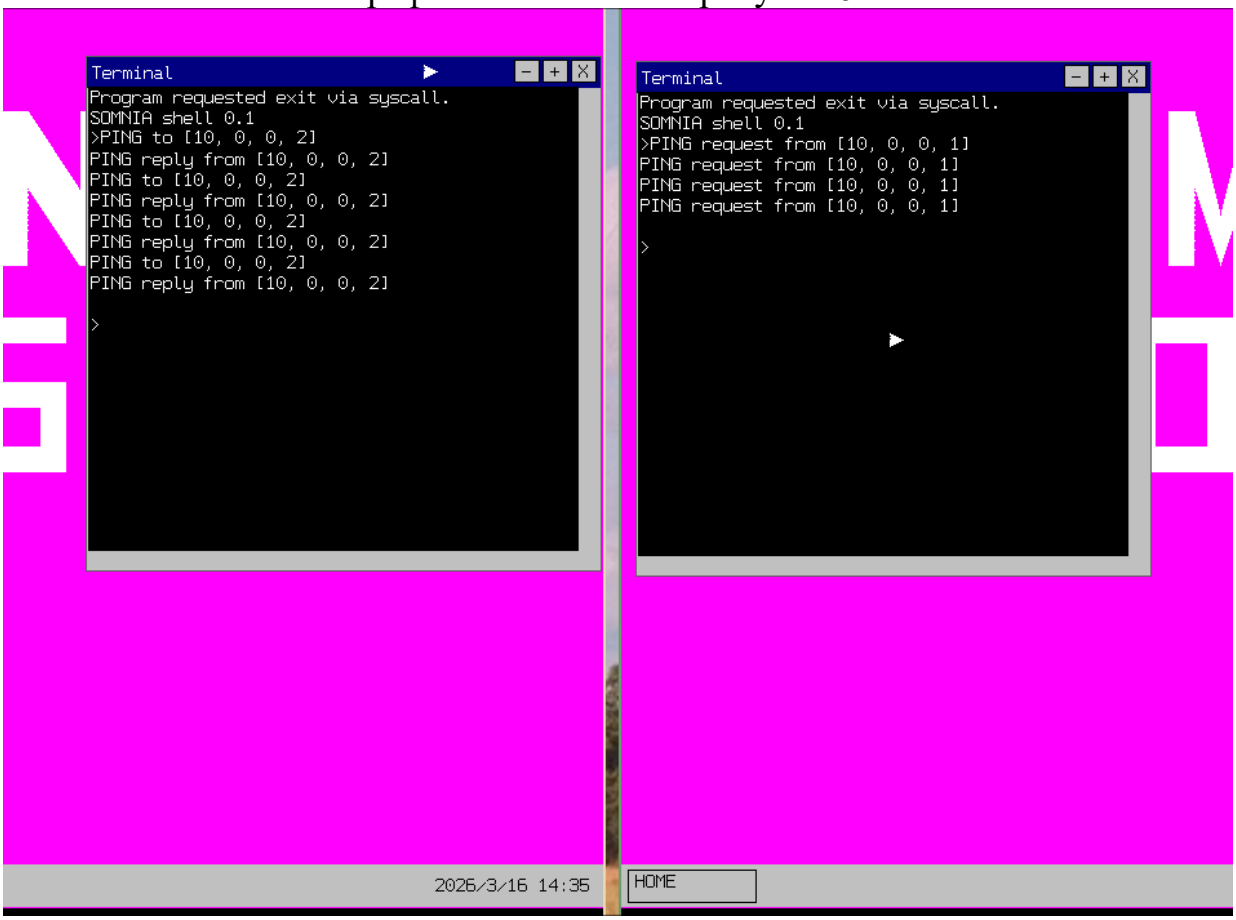
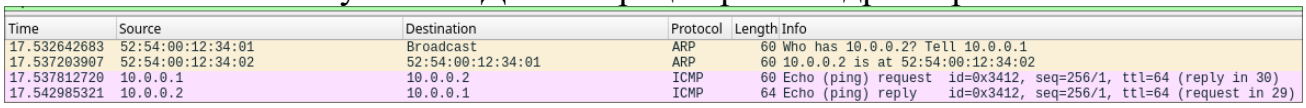


Рисунок 5 – Демонстрация работы драйвера



Time	Source	Destination	Protocol	Length	Info
17.532642683	52:54:00:12:34:01	Broadcast	ARP	60	Who has 10.0.0.2? Tell 10.0.0.1
17.537203907	52:54:00:12:34:02	52:54:00:12:34:01	ARP	60	10.0.0.2 is at 52:54:00:12:34:02
17.537812720	10.0.0.1	10.0.0.2	ICMP	60	Echo (ping) request id=0x3412, seq=256/1, ttl=64 (reply in 30)
17.542985321	10.0.0.2	10.0.0.1	ICMP	64	Echo (ping) reply id=0x3412, seq=256/1, ttl=64 (request in 29)

Рисунок 6 – Некоторые пакеты в сети виртуальных машин, обнаруженные Wireshark

Подводя итог, можно сказать, что реализованный драйвер сетевого контроллера Intel e1000 обеспечивает базовую, но полностью работоспособную сетевую функциональность в рамках поставленной задачи. Несмотря на минималистичный характер реализации, заложенная архитектура может служить основой для дальнейшего развития сетевого стека и добавления поддержки более сложных протоколов.

Список литературы

1. Intel Corporation. PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer’s Manual : 82540EP/EM, 82541xx, 82544GC/EI, 82545GM/EM, 82546GB/EB, and 82547xx. – [Rev. 4.0]. – Intel Corporation, 2009. – 396 p. – (Document Number: 317453006EN). – URL: https://www.intel.com/content/dam/doc/manual/pci-pci-x-family-gbe-controllers-software-dev-manual.pdf (дата обращения: 17.03.2026).

2. Intel 8254x [Электронный ресурс] // OSDev Wiki. – Дата обновления: 30.09.2023. – URL: https://wiki.osdev.org/Intel_8254x (дата обращения: 17.03.2026).

3. Postel, J. Internet Control Message Protocol : RFC 792 [Электронный ресурс] / J. Postel. – ISI, 1981. – 21 p. – URL: https://datatracker.ietf.org/doc/html/rfc792 (дата обращения: 17.03.2026).

4. Ethernet (IEEE 802.3) // Cisco Prime Network 3.8 Reference Guide. Cisco Systems, 2011. URL: https://www.cisco.com/c/en/us/td/docs/net_mgmt/prime/network/3-8/reference/guide/ethrnt.html (дата обращения: 17.03.2026).