Project 5 Device Driver 设计文档

中国科学院大学
贾志杰
2020 年 12 月 28 日

1. 网卡驱动

（1）S-Core 实现时，你初始化了几个接收描述符（RDES），每个接收描述符中设置了哪几个域的值，所设置的域的各自含义是什么？

do_net_recv 函数的参数中要接收几个包就初始化几个接收描述符。设置了 own 为 0，表示主机控制该描述符。first_descriptor 和 last_descriptor 为 1，代表该描述符是该包的第一个也是最后一个描述符，即一个包只用一个 buffer 接收。rx_mac_addr 为 0，不使用校验。disable_ic 为 1，接收完成一个包后不触发中断。second_address_chained 为 1，表示链式描述符结构。end_ring 最后一个描述符为 1 其他为 0，表示环状结构。buffer1_size 设置为缓冲区大小。des2 设置为 buffer 的物理地址，des3 设置为下一个描述符的物理地址。

（2）在 A-Core 实现时，你在哪个函数中判断接收到的数据包数量已经足够了？另外，你的设计中，每接收到几个网络包时会产生一次中断？为什么这么设计？

在 do_net_recv 函数里判断收到的数据包数量是否已经足够。理论上每接收一个网络包都会产生一次中断，但在中断处理函数里会等待本次接收到的所有包都接收完才退出，比如 send 10 会把 10 个包接收完才清除中断。这样设计的原因是 DMA 搬运包和中断处理函数是同时执行的，在第一个包的中断处理函数的执行过程中 DMA 可能已经搬运到第 7、8 个包了，如果这时读取 own 位判定收到 8 个包，在后面清除中断时 DMA 可能已经搬完 10 个包，因此后面 2 个包不会再触发中断了，那就导致统计漏了 2 个包。为了避免统计漏包，在一次中断处理函数中要等到所有包接收完才退出。

（3）DMA 接收和发送描述符采用环形链表和链型链表都是可以的，你认为使用环形链表和使用链型链表有什么区别？

环形便于重复利用描述符，链式在描述符用完后需要手动重新设置所有描述符。

（4）设计或实现过程中遇到的问题和得到的经验

①所谓每接收/发送一个包就要写 DMA 寄存器 2/1 一次，并不是每完成一个包再写一次，可以在最开始连续写目标次数。

②只使用 clear_interrupt 并不能完全清除中断，还需要设置 intenset 和 intenclr 寄存器。

2. C-Core 设计

（1）设计或实现过程中遇到的问题和得到的经验

C core 只需要把分开编译和共享内存的功能用起来即可，需要注意 2 个问题，一是需要把测试文件中收发 2 个函数分成 2 个文件分开编译，二是修正测试代码中一些语法错误和系统调用函数名错误。

3. 关键函数功能

（1）接收描述符初始化函数

```
static void mac_recv_desc_init(mac_t *mac, uint64_t buf_addr, uint64_t num)
{
    int i;
    if(num <= 2)
    num = 3;
    for(i=0; i<num; i++){
        memset(&recv_desc[i], 0, sizeof(dma_desc_t));
        recv_desc[i].des01.erx.own = 0;
        recv_desc[i].des01.erx.first_descriptor = 1;
        recv_desc[i].des01.erx.last_descriptor = 1;
        recv_desc[i].des01.erx.rx_mac_addr = 0;
        recv_desc[i].des01.erx.disable_ic = 0;
        recv_desc[i].des01.erx.second_address_chained = 1;
        recv_desc[i].des01.erx.end_ring = 0;
        recv_desc[i].des01.erx.buffer1_size = PSIZE * 4;
        recv_desc[i].des2 = (uint32_t)((buf_addr + i * PSIZE * 4) & 0x1fffffff);
        recv_desc[i].des3 = (uint32_t)((uint64_t)(&recv_desc[i+1]) & 0x1fffffff);
    }
    recv_desc[num-1].des01.erx.end_ring = 1;
    recv_desc[num-1].des3 = (uint32_t)((uint64_t)(&recv_desc[0]) & 0x1fffffff);
    mac->rd = (uint64_t)recv_desc;
    mac->rd_phy = (uint32_t)((uint64_t)(&recv_desc[0]) & 0x1fffffff);
}
```

（2）发送描述符初始化函数

```
static void mac_send_desc_init(mac_t *mac, uint64_t buf_addr, uint64_t num)
{
    int i;
    if(num <= 2)
    num = 3;
    for(i=0; i<num; i++){
        memset(&send_desc[i], 0, sizeof(dma_desc_t));
        send_desc[i].des01.etx.own = 0;
        send_desc[i].des01.etx.interrupt = 0;
        send_desc[i].des01.etx.last_segment = 1;
        send_desc[i].des01.etx.first_segment = 1;
        send_desc[i].des01.etx.crc_disable = 1;
        send_desc[i].des01.etx.checksum_insertion = 1;
        send_desc[i].des01.etx.end_ring = 0;
        send_desc[i].des01.etx.second_address_chained = 1;
        send_desc[i].des01.etx.buffer1_size = PSIZE * 4;
        send_desc[i].des2 = (uint32_t)((buf_addr) & 0x1fffffff);
        send_desc[i].des3 = (uint32_t)((uint64_t)(&send_desc[i+1]) & 0x1fffffff);
    }
    send_desc[num-1].des01.etx.end_ring = 1;
    send_desc[num-1].des3 = (uint32_t)((uint64_t)(&send_desc[0]) & 0x1fffffff);
    mac->td = (uint64_t)send_desc;
    mac->td_phy = (uint32_t)((uint64_t)(&send_desc[0]) & 0x1fffffff);
}
```

（3）接收包函数

```
uint32_t do_net_recv(recv_arg_t *recv_reg)
{
    mac_t mac;
    uint64_t buf_addr, size, num;
    uint64_t *length;
    buf_addr = recv_reg->buf_addr;
    size = recv_reg->size;
    num = recv_reg->num;
    length = (uint64_t *)(recv_reg->length);

    mac.mac_addr = GMAC_BASE_ADDR;
    mac.dma_addr = DMA_BASE_ADDR;
    mac.psize = PSIZE * 4; // 128bytes
    mac.pnum = PNUM;       // pnum
    mac.daddr = buf_addr;
    mac_recv_desc_init(&mac, buf_addr, num);
    dma_control_init(&mac, DmaStoreAndForward | DmaTxSecondFrame | DmaRxThreshCtrl1
28);
    clear_interrupt(&mac);
    mii_dul_force(&mac);
    reg_write_32(DMA_BASE_ADDR + 0xc, mac.rd_phy);
    reg_write_32(GMAC_BASE_ADDR, reg_read_32(GMAC_BASE_ADDR) | 0x4);
    reg_write_32(DMA_BASE_ADDR + 0x18, reg_read_32(DMA_BASE_ADDR + 0x18) | 0x022000
02); // start tx, rx
    reg_write_32(DMA_BASE_ADDR + 0x1c, 0x10001 | (1 << 6));
    reg_write_32(DMA_BASE_ADDR + 0x1c, DMA_INTR_DEFAULT_MASK);
    int i;
    recv_work = 1;
    for(i=0; i<num; i++){
        recv_desc[i].des01.erx.own = 1;
        length[i] = PSIZE * 4;
    }
    for(i=0; i<num; i++){
        reg_write_32(DMA_BASE_ADDR + 0x8, 1);
    }
    recv_now = 0;
    while(recv_now < num){
        recv_pid = current_running->pid;
        current_running->status = TASK_BLOCKED;
        current_running->inqueue = BLOCK;
        do_scheduler();
    }
    recv_work = 0;
    return 0;
}
```

（4）发送包函数

```
void do_net_send(uint64_t buf_addr, uint64_t size, uint64_t num)
{
    mac_t mac;
    mac.mac_addr = GMAC_BASE_ADDR;
    mac.dma_addr = DMA_BASE_ADDR;
    mac.psize = PSIZE * 4;
    mac.pnum = PNUM;
    mac.saddr = buf_addr;
    mac_send_desc_init(&mac, buf_addr, num);
    dma_control_init(&mac, DmaStoreAndForward | DmaTxSecondFrame | DmaRxThreshCtrl1
28);
    clear_interrupt(&mac);
    mii_dul_force(&mac);
    reg_write_32(DMA_BASE_ADDR + 0x10, mac.td_phy);
    reg_write_32(GMAC_BASE_ADDR, reg_read_32(GMAC_BASE_ADDR) | 0x8);
     // enable MAC-TX
    reg_write_32(DMA_BASE_ADDR + 0x18, reg_read_32(DMA_BASE_ADDR + 0x18) | 0x022020
00); //0x02202002); // start tx, rx
    reg_write_32(DMA_BASE_ADDR + 0x1c, 0x10001 | (1 << 6));
    reg_write_32(DMA_BASE_ADDR + 0x1c, DMA_INTR_DEFAULT_MASK);
    int i;
    for(i=0; i<num; i++){
        send_desc[i].des01.etx.own = 1;
        reg_write_32(DMA_BASE_ADDR + 0x4, 1);
        while(1){
            if(send_desc[i].des01.etx.own == 0)
            break;
        }
    }
}
```

（5）中断使能初始化

```
void register_irq_handler()
{
    volatile uint32_t *intenset_0, *intenclr_0, *intedge_0, *intpol_0, *intauto_0,
*intbounce_0;
    intenset_0 = (volatile uint32_t *)(0xffffffff1fe11428 | 0xa0000000);
    intenclr_0 = (volatile uint32_t *)(0xffffffff1fe1142c | 0xa0000000);
    intedge_0 = (volatile uint32_t *)(0xffffffff1fe11434 | 0xa0000000);
    intpol_0 = (volatile uint32_t *)(0xffffffff1fe11430 | 0xa0000000);
    intauto_0 = (volatile uint32_t *)(0xffffffff1fe1143c | 0xa0000000);
    intbounce_0 = (volatile uint32_t *)(0xffffffff1fe11438 | 0xa0000000);
    *intpol_0 = 0;
    *intauto_0 = 0;
    *intbounce_0 = 0;
    *intenclr_0 = (*intenclr_0) | (1<<12);
    *intenset_0 = (*intenset_0) | (1<<12);
    *intedge_0 = (*intedge_0) | (1<<12);
}
```

（6）注册网卡中断处理函数

```
void interrupt_helper(uint32_t status, uint32_t cause, uint64_t epc)
{
    int IP = (cause & 0xff00) >> 8;
    volatile uint32_t *intisr_0;
    intisr_0 = (volatile uint32_t *)(0xffffffff1fe11420 | 0xa0000000);

    if( (IP & 0x80) ){
        irq_timer();
    }
    else if( (IP & 0x10) ){
        if( (((*intisr_0) & (1<<12)) == (1<<12)) ){
            //printk("mac int");
            mac_irq_handle();
        }
        else{
            printk("[other interrupt] EPC: 0x%x  CAUSE: 0x%x  IP: 0x%x.\n", epc, ca
use, IP);
            while(1);
        }
    }
    else{
        printk("[other interrupt] EPC: 0x%x  CAUSE: 0x%x  IP: 0x%x.\n", epc, cause,
 IP);
        while(1);
    }
}
```

参考文献

[1] MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual, Revision 6.05, 2016
[2] 龙芯 GS264 处理器核用户手册, v1.0, 2018: 93-94, 97-98
[3] 龙芯 2H 处理器用户手册, v1.6, 2016: 151-182