Project 5 Device Driver 设计文档

中国科学院大学
葛忠鑫
2021.1.4

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？

S-Core 实现时，你初始化了 PNUM = 10 个接收描述符（RDES）。初始化接受描述符时初始化了：

- ✧ OWN 位：在初始化接收描述符时可以将 RDES0 的 OWN 位赋值为 0，在开始接收数据（即对 Receive Poll Demand Register 写任意值）时需要将 RDES0 的 OWN 位赋值为 1，表示此接收描述符被 DMA 拥有，此后当 OWN 位变成 0 时，说明此接收描述符已经接收到数据，可以读取接收的数据包。
- ✧ FS（First Desciptor）：为 1 时，表示当前描述符所指向的 buffer 为当前接收帧的第一个保存 buffer。初始化为 1。
- ✧ LS（Last Desciptor）：为 1 时，表示当前描述符所指向的 buffer 为当前接收帧的最后一个保存 buffer。初始化为 1。
- ✧ Extended Status Available/Rx MAC Address：初始化为 0。当高级时间戳 (Advanced Time Stamp)或者 ip 校验去载 (IP Checksum Offload) 功能启用时，该位为 1 表示 RDES4 中记录的扩展状态是有效的。该位仅在 LS(RDES0[8]) 为 1 时有效。当高级时间戳 (Advanced Time Stamp) 和 ip 完全校验去载功能未被启用时 (IPC Full Offload) ，该位表示接收 MAC 地址状态。当该位为 1 时表示接收帧的目的 MAC 地址与 DA(Destination Filter) 寄存器的值匹配，反之为不匹配。
- ✧ DIC（Disable Intr in Completion）：为 1 时表示该帧接收完成后将不会置起 STATUS 寄存器中 RI 位 (CSR5[6])，这将会使得主机无法检测到该中断。在 S-core 中初始化为 1，在实现网卡中断后初始化为 0。
- ✧ RER（Receive End of Ring）：为 1 时表示该描述符为环型描述符链表的最后一个，下一个描述符的地址为接收描述符链的基址。
- ✧ RCH（Second Address Chained）：该位为 1 时表示描述符中的第二个 buffer 地址指向的是下一个描述符的地址，为 0 时表示该地址指向第二个 buffer 地址当该位为 1 时，RDES1[21-11] 的值将没有意义，RDES1[25] 比 RDES1[24] 具有更高优先级（代表环型而不是链型）。初始化为 1。
- ✧ RBS2（Receive Buffer Size 1）：表示数据 buffer1 的大小。根据系统总线的宽度 32/64/128 ，Buffer1 的大小应该为 4/8/16 的整数倍。如果不满足则会导致未知的结果。该域一直有效。如果该域值为 0 ，DMA 则会自动访问 buffer2 或者下一个接收描述符。初始化为 psize。
- ✧ RDES2 域：数据接收 buffer1 的物理地址。
- ✧ RDES3 域：记录了数据接收 buffer2 的物理地址。如果描述符是以链式连接，

则 RDES3 内存储的是下一个接收描述符的物理地址。由于我们采用环形链表，所以初始化为下一个接收描述符的物理地址。

A-core 最终接收描述符初始化代码如下：

```
1.  static void mac_recv_desc_init(mac_t *mac)
2.  {
3.      uint32_t OWN = 0;
4.      uint32_t FS = 1, LS = 1;
5.      uint32_t DIC = 0;
6.      int i;
7.      int num_desc = (mac->pnum > NUM_DMA_DESC) ? NUM_DMA_DESC : mac->pnum;
8.      for(i = 0; i < num_desc; i++) {
9.          rx_descriptor[i].tdes0 = OWN << 31 | FS << 9 | LS << 8 | 0;
10.         rx_descriptor[i].tdes1 = DIC << 31 | (i == num_desc - 1) << 15 | 1
    << 14 | mac->psize;
11.         rx_descriptor[i].tdes2 = mac->daddr_phy + i * mac->psize;
12.         rx_descriptor[i].tdes3 = mac->rd_phy + (((i + 1) % num_desc) * size
    of(desc_t));
13.     }
14. }
```

(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_wait_recv_package 函数中判断接收到的数据包数量已经足够了。do_wait_recv_package 函数如下：

```
1.  void do_wait_recv_package(void)
2.  {
3.      int i;
4.      bzero(recv_flag, 4 * PNUM);
5.      for (i = 0; i < PNUM; i++) {
6.          if (recv_flag[i] == 0) {
7.              do_block(&recv_block_queue);
8.          }
9.      }
10. }
```

每接收到 1 个网络包时就会产生一次中断。因为操作系统可能发一些无效的包，如果每次收到包不加判断，那么就会导致有效包因描述符不够用而丢包。mac_irq_handle 函数如下：

```
1.  void mac_irq_handle(void)
2.  {
3.      volatile uint32_t * intenset_0 = (void *)0xffffffffbfe11428;
4.      volatile uint32_t * intenclr_0 = (void *)0xffffffffbfe1142c;
5.      static uint32_t num_package = 0;
6.      if (num_package == PNUM) {
7.          num_package = 0;
8.      }
```

```
9.      while (!(0x80000000 & rx_descriptor[num_package % NUM_DMA_DESC].tdes0)
    && num_package < PNUM){
10.         recv_flag[num_package] = 1;
11.         if (!queue_is_empty(&recv_block_queue)) {
12.             queue_push(&ready_queue, queue_dequeue(&recv_block_queue));
13.         }
14.         num_package++;
15.     }
16.     *intenclr_0 = (*intenclr_0) | (1 << 12);
17.     *intenset_0 = (*intenset_0) | (1 << 12);
18.     clear_interrupt();
19. }
```

(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？

采用环形链表能够更加方便的重复使用描述符。

在使用单向链表实现描述符时，一轮传输完成后，就需要重新配置相关的 DMA 寄存器和 MAC 寄存器，才可重复利用。但使用循环链表就不需要进行这些操作，只用重新置位描述符 OWN 位和向 DMA 寄存器 1（或 2）写任意值即可。

(4) 设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

2. C-Core 设计

如果有以下内容，请记录

（1）设计或实现过程中遇到的问题和得到的经验（如果有的话可以写下来，不是必需项）

3. 关键函数功能

请列出上述各项功能设计里，你觉得关键的函数，及其作用

(1) do_net_recv 函数

```
1.  /* buf_addr is the total recv buffer's address; size means the total size of
    recv buffer;
2.  the total recv buffer may have some little recv buffer;
3.  num means recv buffer's num ;length is each small recv buffer's size*/
4.  uint32_t do_net_recv(uint64_t buf_addr, uint64_t size, uint64_t num)//, uint6
    4_t length)
5.  {
6.
7.      mac_t mac;
8.
9.      mac.mac_addr = GMAC_BASE_ADDR;
10.     mac.dma_addr = DMA_BASE_ADDR;
11.
12.     mac.psize = PSIZE * 4; // 128bytes
```

```
    mac.pnum = num;       // pnum
    // mac.daddr = buf_addr;
    mac.daddr = (uint64_t)&recv_buf;
    mac.daddr_phy = (uint64_t)(&recv_buf) & 0x1fffffff;
    mac.rd = (uint64_t)&rx_descriptor;
    mac.rd_phy = (uint64_t)(&rx_descriptor) & 0x1fffffff;

    mac_recv_desc_init(&mac);
    dma_control_init(&mac, DmaStoreAndForward | DmaTxSecondFrame | DmaRxThres
hCtrl128);
    clear_interrupt(&mac);

    mii_dul_force(&mac);

    reg_write_32(GMAC_BASE_ADDR, (1 << 8) | 0x4);
    reg_write_32(DMA_BASE_ADDR + DmaRxBaseAddr, (uint32_t)mac.rd_phy);
    reg_write_32(DMA_BASE_ADDR + 0x18, reg_read_32(DMA_BASE_ADDR + 0x18) | 0x
02200002); // start tx, rx
    reg_write_32(DMA_BASE_ADDR + 0x1c, 0x10001 | (1 << 6));

    reg_write_32(DMA_BASE_ADDR + 0x1c, DMA_INTR_DEFAULT_MASK);

    /*  YOU NEED ADD RECV CODE*/
    // do_wait_recv_package();
    int i;
    int OWN = 1;
    for (i = 0; i < mac.pnum; i++) {
        rx_descriptor[i].tdes0 |= OWN << 31;
        reg_write_32(DMA_BASE_ADDR + DmaRxPollDemand, 0x00000001);
    }
    mac_recv_handle(&mac);
    memcpy(buf_addr, &recv_buf, size * 4);
    // kprintf_recv_buffer(buf_addr);
    return 0;
}
```

(2) do_net_send 函数

```
/* buf_addr is sended buffer's address; size means sended buffer's size;
num means sended buffer's times*/
void do_net_send(uint64_t buf_addr, uint64_t size, uint64_t num)
{
uint64_t td_phy;
mac_t mac;
mac.mac_addr = GMAC_BASE_ADDR;
mac.dma_addr = DMA_BASE_ADDR;
```

```
9.  memcpy(&send_buf, buf_addr, size * 4);

10. mac.psize = size * 4;
11. mac.pnum = num;
12. // td_phy = mac.td_phy;
13. mac.saddr = (uint64_t)&send_buf;
14. mac.saddr_phy = (uint64_t)(&send_buf) & 0x1fffffff;
15. mac.td = (uint64_t)&tx_descriptor;
16. mac.td_phy = (uint64_t)(&tx_descriptor) & 0x1fffffff;

17. mac_send_desc_init(&mac);
18. dma_control_init(&mac, DmaStoreAndForward | DmaTxSecondFrame | DmaRxThreshCtr
    l128);
19. clear_interrupt(&mac);

20. mii_dul_force(&mac);

21. reg_write_32(GMAC_BASE_ADDR, (1 << 8) | 0x8);                    // enable MA
    C-TX
22. // reg_write_32(GMAC_BASE_ADDR, reg_read_32(GMAC_BASE_ADDR) | 0x8);
              // enable MAC-TX
23. reg_write_32(DMA_BASE_ADDR + DmaTxBaseAddr, (uint32_t)mac.td_phy);
24. reg_write_32(DMA_BASE_ADDR + 0x18, reg_read_32(DMA_BASE_ADDR + 0x18) | 0x0220
    2000); //0x02202002); // start tx, rx
25. reg_write_32(DMA_BASE_ADDR + 0x1c, 0x10001 | (1 << 6));
26. reg_write_32(DMA_BASE_ADDR + 0x1c, DMA_INTR_DEFAULT_MASK);

27. /*  YOU NEED ADD SEND CODE*/
28. int i;
29. int OWN = 1;
30. for (i = 0; i < mac.pnum; i++) {
31. tx_descriptor[i].tdes0 |= OWN << 31;
32. reg_write_32(DMA_BASE_ADDR + DmaTxPollDemand, 0x00000001);
33. }
34. for (i = 0; i < mac.pnum; i++) {
35. while(0x80000000 & tx_descriptor[i].tdes0);
36. }
37. }
```

参考文献

[1] Loongson_2H_cpu_user