Bios-dyh 中 TFTP 的总体分析

杜云海（duyunhai@hotmail.com ）
2004 年 6 月

想分析 tftp 协议是觉得这个协议非常适合用于嵌入式系统中，不仅局限于 bios，也许将来其他场合也有可能使用这个协议，弄懂它，一劳永逸。

TFTP 协议简介

TFTP（Trivial File Transfer Protocol），可以翻译为“简单文件传输协议”。

TFTP 协议最初打算用于引导无盘系统，TFTP 的代码（和它所需要的 UDP、IP 和设备驱动程序）能适合只读存储器。它比 FTP 简单也比 FTP 功能少。它在不需要用户权限或目录可见的情况下使用。它使用 UDP 协议而不是 TCP 协议，它在 RFC 1350 内得到详细说明。

下图显示了 5 种 TFTP 报文格式（操作码为 1 和 2 的报文使用相同的格式）。

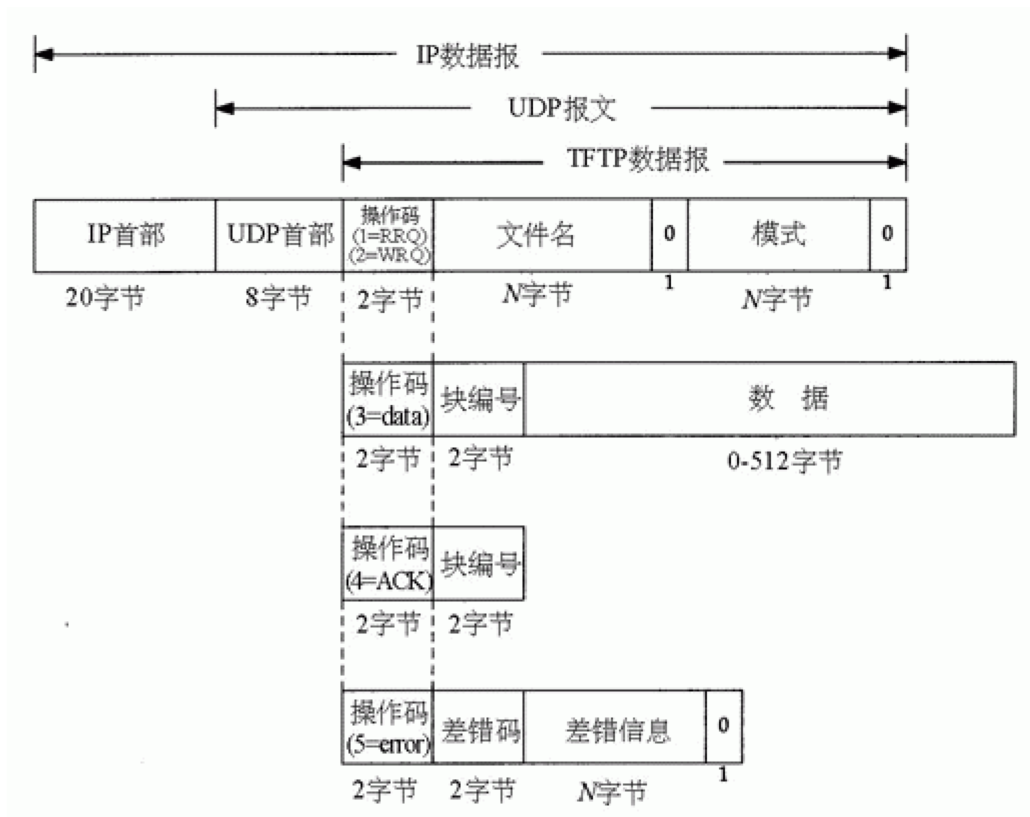

图 1　5 种 TFTP 报文格式

TFTP 报文的头两个字节表示操作码。对于读请求（RRQ）和写请求（WRQ），文件名字段说明客户要读或写的位于服务器上的文件。这个文件字段以 0 字节作为结束。模式字段是一个 ASCII 码串 netascii 或 octet（可大小写任意组合），同样以 0 字节结束。netascii 表示数据是以成行的 ASCII 码字符组成，以两个字节—回车字符后跟换行字符（称为 CR/LF）作为行结束符。这两个行结束字符在这种格式和本地主机使用的行定界符之间进行转化。octet

则将数据看作 8bit 一组的字节流而不作任何解释。

每个数据分组包含一个块编号字段，它以后要在确认分组中使用。以读一个文件作为例子，TFTP 客户需要发送一个读请求说明要读的文件名和文件模式(mode)。如果这个文件能被这个客户读取，TFTP 服务器就返回一个块编号为 1 的数据分组。TFTP 客户又发送一个块编号为 1 的 ACK。TFTP 服务器随后发送块编号为 2 的数据。TFTP 客户发回块编号为 2 的 ACK。重复这个过程直到这个文件传送完。除了最后一个数据分组可含有不足 512 字节的数据，其他每个数据分组均含有 512 字节的数据。当 TFTP 客户收到一个不足 512 字节的数据分组，就知道它收到最后一个数据分组。

在写请求的情况下，TFTP 客户发送 WRQ 指明文件名和模式。如果该文件能被该客户写，TFTP 服务器就返回块编号为 0 的 A C K 包。该客户就将文件的头 512 字节以块编号为 1 发出。服务器则返回块编号为 1 的 ACK。

这种类型的数据传输称为“ 停止等待协议 ”。它只用在一些简单的协议如 TFTP 中。TFTP 的优点在于实现的简单而不是高的系统吞吐量。

在 bios-dyh 中，经过根目录下 bios.c 中的 int load_tftp(unsigned long mode, unsigned long param)进入 tftp 文件下的 main.c，同时传递两个参数，作为全局变量。一系列初始化后，系统进入一个 while 循环，即进入等待客户请求以及数据的 TFTP 服务器状态。

\bios-dyh1.4\tftp\main.c

```
int startup_mode;
int startup_param;

    eth_init();
    eth_get_addr(eth_addr);
    arp_init();
    ip_init(ip);
    udp_init();
    arp_add_entry(eth_addr, ip);

    while (1) {
        net_handle();
        timeout_handle();
        if (kbhit() && (getch() == KEY_ESC))
            break;
    }
```

所有这些初始化过程本文暂时不予关心，否则就要牵涉到太多有关硬件的问题。我们大概知道初始化后系统建立了一个足够运行 TFTP 协议的 eth→ip→udp 网络协议栈，当然其中要用到一些数据结构，后面会陆续引用并分析。

其中核心数据结构是 struct sk_buff *skb，这个结构是网络协议栈实现以及分析的主要结构体，并贯穿其中，大多数网络协议栈也是这么解析的。

\bios-dyh1.4\tftp\skbuff.h

```
struct sk_buff {
    unsigned char pad[2];
    unsigned char buf[ETH_FRAME_LEN];
    unsigned int truesize;          /* Buffer size                    */
    unsigned char *data;            /* Data head pointer                  */
    unsigned int len;           /* Length of actual data            */
};
```

进入了 tftp 服务器最重要，也是唯一的一个处理函数：

\bios-dyh1.4\tftp\main.c

```
int net_handle(void)
{
    struct sk_buff *skb;
    struct ethhdr *eth_hdr;

    skb = alloc_skb(ETH_FRAME_LEN);

    if (eth_rcv(skb) != -1) {

        eth_hdr = (struct ethhdr *)(skb->data);
        skb_pull(skb, ETH_HLEN);

        if (ntohs(eth_hdr->h_proto) == ETH_P_ARP)
            arp_rcv_packet(skb);

        else if(ntohs(eth_hdr->h_proto) == ETH_P_IP)
            ip_rcv_packet(skb);
    }

    free_skb(skb);

    return 0;
}
```

如上代码所示，首先分配一个 sk_buff 结构的缓冲区 skb，然后调用 eth_rcv(skb)从底层硬件接收 ETH 以太网帧到 skb，然后对 skb 中的数据区 data 进行解析，应用 ethhdr 强制类型转换得到一个以太网帧头，然后通过 skb_pull 函数去掉以太网帧头得到一个 IP 数据报，最后通过判断以太网帧的 eth_hdr->h_proto 调用 arp_rcv_packet(skb) 或 ip_rcv_packet(skb)，从而将帧送到上一层分析。

其中 MAC 帧和 ETH 帧的接收主要代码在 mac.c 和 eth.c 中，本文这里就不赘述了。

arp_rcv_packet(skb)——ARP 数据包这里也不是分析的重点。

我们主要来分析 ip_rcv_packet(skb)，其中 skb 已经经过处理，不再是 ETH 帧，而是 IP 数据报。

\bios-dyh1.4\tftp\ip.c

```
int ip_rcv_packet(struct sk_buff *skb)
{
    struct iphdr *ip_hdr = (struct iphdr *)(skb->data);

    if (ntohl(ip_hdr->daddr) == local_ip)
    {
        skb->len = ntohs(ip_hdr->tot_len);
        skb_pull(skb, sizeof(struct iphdr));

        if (ip_hdr->protocol == UDP)
            udp_rcv_packet(skb);
    }

    return 0;
}
```

故伎重施,还是对 skb 进行相应的强制转换,得到 IP 帧头,当然其中数据位置都是根据 IETF 协议定义好的了。

然后根据 IP 帧头进行 IP 地址比较，以及得到 IP 帧头的长度，最后再通过 skb_pull () 得到 UDP 报文。TFTP 是使用 UDP 协议而不是 TCP 协议。

处理 UDP 报文得到 TFTP 数据报，udp_rcv_packet(skb)和上面的处理过程基本相同。

\bios-dyh1.4\tftp\udp.c

```
int udp_rcv_packet(struct sk_buff *skb)
{
    struct udphdr *udp_hdr = (struct udphdr *)(skb->data);

    skb->len = ntohs(udp_hdr->len);
    skb_pull(skb, sizeof(struct udphdr));

    if (ntohs(udp_hdr->dest) == TFTP)
        tftp_rcv_packet(skb);

    return 0;
}
```

以上这些过程只是得到一个 UDP 报文，和 TFTP 协议的具体实现没有直接的关系，也就是说即使 TFTP 没有实现，UDP 报文一样会产生。下面我们就来看看 TFTP 的具体代码实现。

TFTP 协议实现

由于 TFTP 使用的是停止等待协议，所以应该说他的代码是比较容易看懂的，起码比起 TCP 协议来说。而且其数据报文的格式也比较简单。虽然简单，但是用在嵌入式系统中，却令人感觉非常实用。

下面我们就开始一步一步分析：

\bios-dyh1.4\tftp\tftp.c

```
int tftp_rcv_packet(struct sk_buff *skb)
{
    struct tftphdr *tftp_hdr;

    tftp_hdr = (struct tftphdr *)skb->data;

    switch (ntohs(tftp_hdr->th_opcode)) {

    case RRQ:
        break;
    case WRQ:
        tftp_rcv_wrq(skb);
        break;
    case DATA:
        tftp_rcv_data(skb);
        break;
    case ACK:
        break;
    case ERROR:
        break;
    default:
        break;
    }

    return 0;
}
```

首先得到 tftp 数据报的头部，然后判断其中的操作码（参考图 1）th_opcode，共有五种：RRQ、WRQ、DATA、ACK、ERROR，由于在 bios-dyh 重我们只需要客户机写数据到服务器（开发板）上，所以只实现了 WRQ 和 DATA 这两种报文格式，如果大家还需要其他报文，可以在这里添加。

现在我们以 bios-dyh 在开发板情况上运行为例

首先开发板上运行的 TFTP 服务器一直等待客户写数据，当我们在 windows2000/XP 客户机的 tftp 客户端键入：

tftp –i <ip address> put <source>

客户端根据 TFTP 的协议，首先发送一个 WRQ 报文给服务器（写请求报文）具体见 TFTP 协议简介。所以服务器首先接收到一个 WRQ 报文：

\bios-dyh1.4\tftp\tftp.c

```
int tftp_rcv_wrq(struct sk_buff *skb)
{
    struct tftphdr *tftp_hdr;

    client_ip = ip_get_source_ip(skb);
    client_port = udp_get_source_port(skb);

    tftp_hdr = (struct tftphdr *)skb->data;
    tftp_send_ack(tftp_hdr, 0);
    client_block = 1;

    tftp_put_begin();

    return 0;
}
```

首先根据 skb 得到客户端的 IP 地址 client_ip 和端口号 client_port，接着得到 UDP 数据报头部，然后按照 TFTP 协议发回 ACK 0，并将块编码 client_block 设为 1，以进行下一步的数据接收判断。

tftp_put_begin()函数主要是在数据传输开始前根据选择项打印相应的信息,并设置一些变量：

\bios-dyh1.4\tftp\tftpput.c

```
int tftp_put_begin(void)
{
    switch (startup_mode) {
    case BOOT_LOAD_PROGRAM:
    case BOOT_UPDATE_FIRMWARE:
    case BOOT_UPDATE_BIOS:
        printf("Starting the TFTP download...\r\n");
        buf = (unsigned char *)0x00008000;
        data_len = 0;
        break;
    default:
        break;
    }
    return 0;
}
```

这里用到了 startup_mode 这个全局变量,这个变量主要是传递 bios-dyh 启动时用户选择的功能,从而在 tftp_put_end()中采用相应的功能函数,而在 tftp 传入数据时没有什么作用。

所有选项都执行相同的代码：

```
        printf("Starting the TFTP download...\r\n");
        buf = (unsigned char *)0x00008000;
        data_len = 0;
```

打印开始头字符，设置 buf 指针为 0x00008000，data_len 接收的数据长度为 0 。这两个变量也是全局变量，在后面的接收数据报函数中使用到。

当服务器发回 ACK 0 给客户端，那么客户端就开始发送数据包（从 BLOCK 1 开始），TFTP 每个数据分组为 512 字节，不满 512 字节的就是最后一个数据分组了。这是服务器端则调用 tftp_rcv_packet() → tftp_rcv_data()。

\bios-dyh1.4\tftp\tftp.c

```
int tftp_rcv_data(struct sk_buff *skb)
{
    struct tftphdr *tftp_hdr;
    int len;

    if (client_ip != ip_get_source_ip(skb))
        return -1;
    if (client_port != udp_get_source_port(skb))
        return -1;

    tftp_hdr = (struct tftphdr *)skb->data;
    if (client_block == ntohs(tftp_hdr->th_block)) {

        len = skb->len - sizeof(struct tftphdr);
        tftp_put(tftp_hdr->th_data, len);

        tftp_send_ack(tftp_hdr, client_block);
        client_block++;

        if (len < 512)
            tftp_put_end();

    } else if (client_block > ntohs(tftp_hdr->th_block)) {

        tftp_send_ack(tftp_hdr, ntohs(tftp_hdr->th_block));

    } else {
```

```
        tftp_send_ack(tftp_hdr, client_block);
    }

    return 0;
}
```

这个是 tftp 协议实现的核心。

首先验证该数据包的 IP 地址和端口号和刚才发送写请求报文是否是同一个客户机进程，然后得到 TFTP 报文头部，接着判断 BLOCK 块编码是否正确，以保证报文的顺序

然后是最重要的一步：tftp_put(tftp_hdr->th_data, len)，将 TFTP 数据报文中的数据部分及其长度送入 tftp_put()进行最真正的解析处理。接着发送该数据块的 ACK 返回给客户机，然后 client_block++。假如数据报文长度小于 512，说明该报文为最后一个报文，所以调用 tftp_put_end()进行最后处理。

假如 client_block 不正确，那么发送正确的 ACK 编码给客户机，要求其重发或其他处理，tftp 就是依靠 ACK 和 BLOCK 的编码来保证协议数据报的工作正常。

然后下一个数据报又重复刚刚的过程……

下面我们来看看 tftp_put()和 tftp_put_end()这两个数据报文核心处理函数，传递的参数为真正的数据和数据长度。

\bios-dyh1.4\tftp\tftpput.c

```
int tftp_put(unsigned char *data, int len)
{
    static int count = 0;

    count += len;
    if (count > 32 * 1024) {
        printf(".");
        count = 0;
    }

    switch (startup_mode) {
    case BOOT_LOAD_PROGRAM:
    case BOOT_UPDATE_FIRMWARE:
    case BOOT_UPDATE_BIOS:
        memcpy(buf + data_len, data, len);
        data_len += len;
        break;
```

```
    default:
        break;
    }
    return 0;
}
```

该函数首先是打印出类似于进度条的点，以指示用户数据正在写入服务器中，每 32K 的数据打印一个点，一般 uClinux 的映象文件 image.ram 为 1.6M，大概打印 50 多个点吧☺

然后再根据 startup_mode 来判断如何处理该数据，暂时所有选项还是直接将数据复制到内存相应区域，其中全局变量 data_len 我们在 tftp_put_begin()中提到过，主要用来记录当前数据总传输的长度，当然这个变量也可以用来指示当前数据 data 缓冲区的尾部。

当接收的数据报文长度小于 512，说明是最后一个报文，则调用 tftp_put_end()

\bios-dyh1.4\tftp\tftpput.c

```
int tftp_put_end(void)
{
    int ch;
    unsigned long l;

    printf("\r\n");

    switch (startup_mode) {
    case BOOT_LOAD_PROGRAM:
        asm(    "
                ldr     r14, =0x00008000;
                mov     PC, r14; "
        );//run uClinux
        break;

    case BOOT_UPDATE_FIRMWARE:
        update_firmware(buf, data_len);
        break;

    case BOOT_UPDATE_BIOS:
        update_bios(buf, data_len);
        break;

    default:
        break;
    }

    printf("Reboot? (y/n) ");
```

```
    while (1) {
        ch = getch();
        if (ch == 'y' || ch == 'Y') {
            putch(ch);
            bios_reboot();
        }
        if (ch == 'n' || ch == 'N') {
            putch(ch);
            break;
        }
    }
    printf("\r\n\r\n");

    return 0;
}
```

从以上 switch 语句可以看出，利用 TFTP 将所有数据传送到相应的缓冲区后，接下来就是根据用户不同的选项 startup_mode 进行不同的‘动作’了。在 bios-dyh1.4 版本中,其中 flash 烧写和 bios 升级暂时没有实现。

- BOOT_LOAD_PROGRAM:启动送入的 uClinux 映象文件；
- 进行 flash 烧写：update_firmware 函数为空
- bios 升级： update_bios 函数为空。

然后询问是否重启系统，于是 bootlaoder 的相应功能也随之实现，TFTP 也完成了其使命。

配置：

1.IP 地址配置：

\bios-dyh1.4\imgtools\param\param.c

tftp_ipaddr: 0xd34156bb, //211.65.86.187

将 IP 地址一一对应地改为 16 进制数即可

2.串口传输速率：

\bios-dyh1.4\console.c

```
int console_init(void)
{
    outl(0x03, ULCON0);
    outl(0x09, UCON0);
//  outl(0x1A0, UBRDIV0);    //57600
    outl(0x500, UBRDIV0);    //19200

    return 0;
}
```

修改其中的 outl(0x500，UBRDIV0)；　//19200　语句，对 UBRDIV0 设置相应值，以得到需要的串口速率，不过最好还是用 19200。

附：bios-dyh1.4——基于“4510B+uClinux 系统”最容易学习的 Bootloader !!

＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝
文章版权所有，请勿随意修改发布本文
转载请注明作者 杜云海（duyunhai@hotmial.com）及网站 希加科技（www.seajia.com）