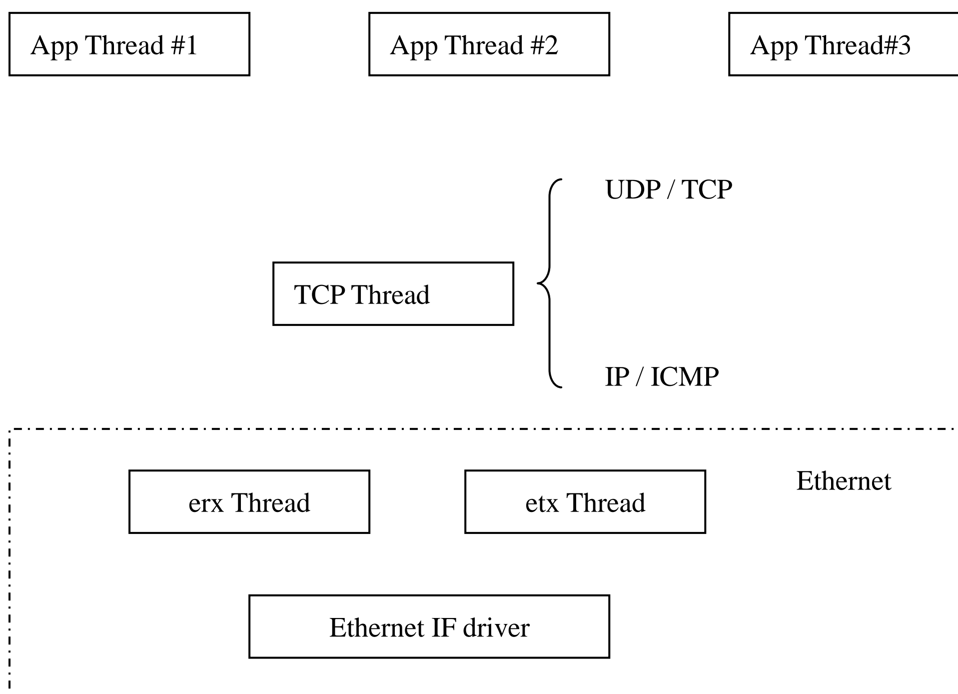
1 RT-Thread 上的 LwIP

1.1 LwIP 简介

LwIP 是瑞士计算机科学院（Swedish Institute of Computer Science）的 Adam Dunkels 等开发的一套用于嵌入式系统的开放源代码 TCP/IP 协议栈，它在包含完整的 TCP 协议实现基础上实现了小型的资源占用，因此它十分适合于使用到嵌入式设备中，占用的体积大概在几十 kB RAM 和 40KB ROM 代码左右。

由于 LwIP 出色的小巧实现，而功能也相对完善（包含相对完整的 BSD 风格 socket 编程），用户群比较广泛。实时线程操作系统（RT-Thread）采用 LwIP 做为默认的 TCP/IP 协议栈，同时根据小型设备的特点对其进行再优化，体积相对进一步减小，**RAM 占用缩小到 5kB 附近**（依据上层应用使用情况会有所浮动）。

RT-Thread 上 LwIP 的模块层次图



TCP Thread 部分是 LwIP 的主线程，各个应用线程通过 LwIP 的接口与 LwIP 线程进行通信（一般采用 MailBox 方式）。而在 RT-Thread 系统中，如果网络物理层是以太网，那么会有两个线程存在：erx 和 etx 分别对应以太网的收发。

1.2 用户数据包协议（UDP）

用户数据包协议（User Datagram Protocol，UDP）是一个无连接协议，传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快地

把它扔到网络上。在发送端，UDP 传送数据的速度仅仅是受应用程序生成数据的速度、网络接口传输速度和传输带宽的限制；在接收端，UDP 把每个消息段放在队列中，应用程序每次从队列中读一个消息段。

由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息。

UDP 信息包的标题很短，只有 8 个字节，相对于 TCP 的 20 个字节信息包的额外开销小很多。

1.3 传输控制协议（TCP）

传输控制协议（Transmission Control Protocol，TCP）是一种面向连接（连接导向）的、可靠的、基于字节流的运输层（Transport layer）通信协议。在简化的计算机网络 OSI 模型中，它完成第四层传输层所指定的功能。

在因特网协议族（Internet protocol suite）中，TCP 层是位于 IP 层之上，应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像管道一样的连接，但是 IP 层不提供这样的流机制，而是提供不可靠的包交换。

应用层向 TCP 层发送用于网间传输的、用 8 位字节表示的数据流，然后 TCP 把数据流分割成适当长度的报文段(通常受该计算机连接的网络的数据链路层的最大传送单元(MTU)的限制)。之后 TCP 把结果包传给 IP 层，由它来通过网络将包传送给接收端实体的 TCP 层。TCP 为了保证不发生丢包，就给每个字节一个序号，同时序号也保证了传送到接收端实体的包的按序接收。然后接收端实体对已成功收到的字节发回一个相应的确认(ACK)； 如果发送端实体在合理的往返时延(RTT)内未收到确认，那么对应的数据（假设丢失了）将会被重传。TCP 用一个校验和函数来检验数据是否有错误；在发送和接收时都要计算校验和。

首先，TCP 建立连接之后，通信双方都同时可以进行数据的传输，其次，他是全双工的；在保证可靠性上，采用超时重传和捎带确认机制。

在流量控制上，采用滑动窗口协议，协议中规定，对于窗口内未经确认的分组需要重传。

在拥塞控制上，采用慢启动算法。

尽管 TCP 和 UDP 都使用相同的网络层（IP），TCP 却向应用层提供与 UDP 完全不同的服务。

TCP 提供一种面向连接的、可靠的字节流服务。

面向连接意味着两个使用 TCP 的应用（通常是一个客户和一个服务器）在彼此交换数据之前必须先建立一个 TCP 连接。这一过程与打电话很相似，先拨号振铃，等待对方摘机

说“喂”，然后才说明是谁。

在一个 TCP 连接中，仅有两方进行彼此通信。广播和多播不能用于 TCP。

1.4 在 STM32 上使用 RT-Thread/LwIP

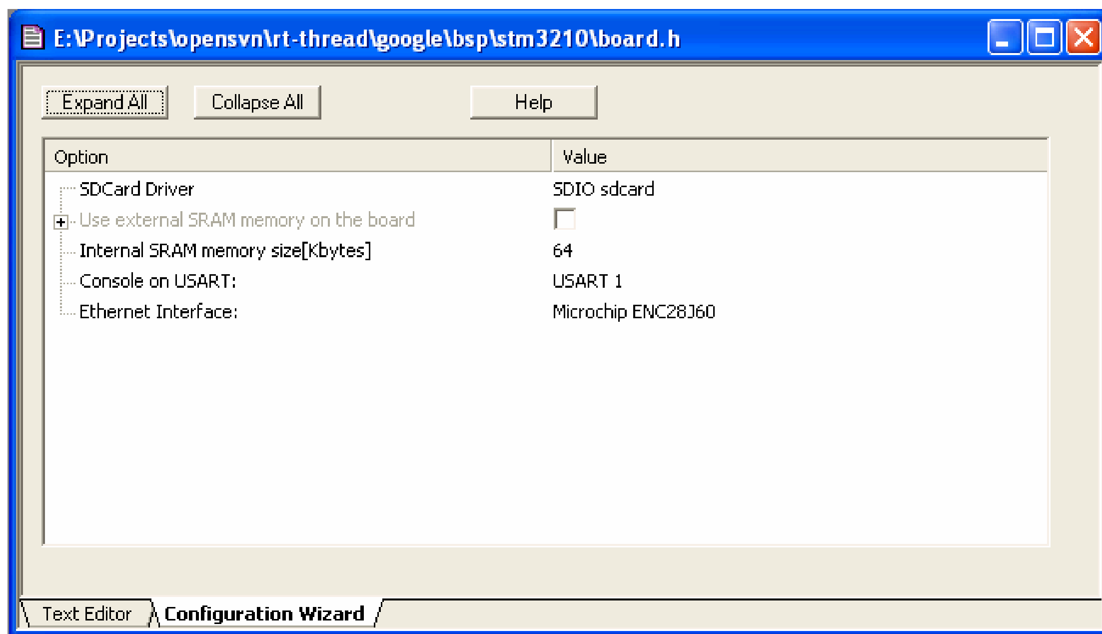
目前在 STM32 上内建了几种以太网驱动，可根据自己硬件情况选择相应驱动：

- Microchip ENC28j60
- DM9000A （16bit 模式）
- STM32F107

如果使用的是前两种驱动，可以选择使用 project_lwip 目录中的工程，把它们都复制到 bsp\stm3210 目录中（重名的文件覆盖即可）。project_lwip 工程的配置是：
RT-Thread Kernel + LwIP，选择 ENC28J60 或 DM9000A 做为以太网接口。

如果是 ENC28J60，其硬件连接是
SPI2 连接 ENC28J60，GPIO B Port12 做为片选，GPIO B Port0 做为触发中断。

DM9000A 只能通过 FSMC 连接到 STM32 上，默认是 STM32F103ZE 芯片，16bit 模式连接到 FSMC Bank1 的 NorSRAM4 上，基地址是 0x6C000000。



Keil MDK 编译后体积指标：
Program Size: Code=50660 RO-data=2388 RW-data=340 ZI-data=6628

LwIP 可支持几种 API 进行网络编程，有原始的接口，面向 netcon、netbuf 的接口，还有 BSD socket 方式的接口。在 RT-Thread 中推荐使用 BSD socket 的接口进行编程，因为 BSD socket

是标准接口，在 Windows、Linux 上都通用，使用 socket 进行编程也可以先在 PC 上调试完毕后再移植到 RT-Thread/LwIP 上。

1.5 RT-Thread/LwIP 上的 TCP 例程

1.5.1 TCP 服务端

以下是如何在 RT-Thread 上使用 BSD socket 接口的一个 TCP 服务端例子，当把这个代码加入到 RT-Thread 时，它会自动向 finsh 命令行添加一个 tcpserv 命令，在 finsh 上执行 tcpserv() 函数即可启动这个 TCP 服务端，它是在端口 5000 上进行监听。

当有 TCP 客户向这个服务端进行连接后，只要服务端接收到数据，它立即向客户端发送“This is TCP Server from RT-Thread.”的字符串。

如果服务端接收到 q 或 Q 字符串时，服务器将主动关闭这个 TCP 连接。如果服务端接收到 exit 字符串时，服务端将退出服务。

例子代码如下：

```
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用BSD Socket接口必须包含sockets.h这个头文件 */

static const char send_data[] = "This is TCP Server from RT-Thread."; /* 
发送用到的数据 */
void tcpserv(void* parameter)
{
   char *recv_data; /* 用于接收的指针，后面会做一次动态分配以请求可用内存 */
   rt_uint32_t sin_size;
   int sock, connected, bytes_received;
   struct sockaddr_in server_addr, client_addr;
   rt_bool_t stop = RT_FALSE; /* 停止标志 */

   recv_data = rt_malloc(1024); /* 分配接收用的数据缓冲 */
   if (recv_data == RT_NULL)
   {
       rt_kprintf("No memory\n");
       return;
   }

   /* 一个socket在使用前，需要预先创建出来，指定SOCK_STREAM为TCP的socket */
   if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
   {
       /* 创建失败的错误处理 */
       rt_kprintf("Socket error\n");
```

```
        /* 释放已分配的接收缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 初始化服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(5000); /* 服务端工作的端口 */
    server_addr.sin_addr.s_addr = INADDR_ANY;
    rt_memset(&(server_addr.sin_zero),8, sizeof(server_addr.sin_zero));

    /* 绑定socket到服务端地址 */
    if (bind(sock, (struct sockaddr *)&server_addr, sizeof(struct
sockaddr)) == -1)
    {
        /* 绑定失败 */
        rt_kprintf("Unable to bind\n");

        /* 释放已分配的接收缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 在socket上进行监听 */
    if (listen(sock, 5) == -1)
    {
        rt_kprintf("Listen error\n");

        /* release recv buffer */
        rt_free(recv_data);
        return;
    }

    rt_kprintf("\nTCPServer Waiting for client on port 5000...\n");
    while(stop != RT_TRUE)
    {
        sin_size = sizeof(struct sockaddr_in);

        /* 接受一个客户端连接socket的请求，这个函数调用是阻塞式的 */
        connected = accept(sock, (struct sockaddr *)&client_addr,
&sin_size);
        /* 返回的是连接成功的socket */
```

```
        /* 接受返回的client_addr指向了客户端的地址信息 */
        rt_kprintf("I got a connection from (%s , %d)\n",

inet_ntoa(client_addr.sin_addr),ntohs(client_addr.sin_port));

        /* 客户端连接的处理 */
        while (1)
        {
            /* 发送数据到connected socket */
            send(connected, send_data, strlen(send_data), 0);

            /* 从connected socket中接收数据，接收buffer是1024大小，但并不一定能够收到
1024大小的数据 */
            bytes_received = recv(connected,recv_data, 1024, 0);
            if (bytes_received < 0)
            {
                /* 接收失败，关闭这个connected socket */
                lwip_close(connected);
                break;
            }

            /* 有接收到数据，把末端清零 */
            recv_data[bytes_received] = '\0';
            if (strcmp(recv_data , "q") == 0 || strcmp(recv_data , "Q") ==
0)
            {
                /* 如果是首字母是q或Q，关闭这个连接 */
                lwip_close(connected);
                break;
            }
            else if (strcmp(recv_data, "exit") == 0)
            {
                /* 如果接收的是exit，则关闭整个服务端 */
                lwip_close(connected);
                stop = RT_TRUE;
                break;
            }
            else
            {
                /* 在控制终端显示收到的数据 */
                rt_kprintf("RECIEVED DATA = %s \n" , recv_data);
            }
        }
    }
```

```
    /* 退出服务 */
    lwip_close(sock);

    /* 释放接收缓冲 */
    rt_free(recv_data);

    return ;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出tcpserv函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(tcpserv, startup tcp server);
#endif
```

1.5.2 TCP 客户端

以下是如何在 RT-Thread 上使用 BSD socket 接口的一个 TCP 客户端例子。当把这个代码加入到 RT-Thread 时，它会自动向 finsh 命令行添加一个 tcpclient 命令，在 finsh 上执行 tcpclient(url, port)函数即可启动这个 TCP 服务端，url 指定了这个客户端连接到的服务端地址或域名，port 是相应的端口号。

当 TCP 客户端连接成功时，它会接收服务端传过来的数据。当有数据接收到时，如果是以 q 或 Q 开头，它将主动断开这个连接；否则会把接收的数据在控制终端中打印出来，然后发送“This is TCP Client from RT-Thread.”的字符串。

例子代码如下：

```
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含netdb.h头文件 */
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

static const char send_data[] = "This is TCP Client from RT-Thread."; /*
发送用到的数据 */
void tcpclient(const char* url, int port)
{
    char *recv_data;
    struct hostent *host;
    int sock, bytes_received;
    struct sockaddr_in server_addr;
```

```
    /* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
    host = gethostbyname(url);

    /* 分配用于存放接收数据的缓冲 */
    recv_data = rt_malloc(1024);
    if (recv_data == RT_NULL)
    {
        rt_kprintf("No memory\n");
        return;
    }

    /* 创建一个socket，类型是SOCKET_STREAM，TCP类型 */
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        /* 创建socket失败 */
        rt_kprintf("Socket error\n");

        /* 释放接收缓冲 */
        rt_free(recv_data);
        return;
    }

    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    rt_memset(&(server_addr.sin_zero), 0,
sizeof(server_addr.sin_zero));

    /* 连接到服务端 */
    if (connect(sock, (struct sockaddr *)&server_addr, sizeof(struct
sockaddr)) == -1)
    {
        /* 连接失败 */
        rt_kprintf("Connect error\n");

        /*释放接收缓冲 */
        rt_free(recv_data);
        return;
    }

    while(1)
    {
        /* 从sock连接中接收最大1024字节数据 */
```

```
        bytes_received = recv(sock, recv_data, 1024, 0);
        if (bytes_received < 0)
        {
            /* 接收失败，关闭这个连接 */
            lwip_close(sock);

            /* 释放接收缓冲 */
            rt_free(recv_data);
            break;
        }

        /* 有接收到数据，把末端清零 */
        recv_data[bytes_received] = '\0';

        if (strcmp(recv_data , "q") == 0 || strcmp(recv_data , "Q") == 0)
        {
            /* 如果是首字母是q或Q，关闭这个连接 */
            lwip_close(sock);

            /* 释放接收缓冲 */
            rt_free(recv_data);
            break;
        }
        else
        {
            /* 在控制终端显示收到的数据 */
            rt_kprintf("\nRecieved data = %s " , recv_data);
        }

        /* 发送数据到sock连接 */
        send(sock,send_data,strlen(send_data), 0);
    }

    return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出tcpclient函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(tcpclient, startup tcp client);
#endif
```

1.6 RT-Thread/LwIP 上的 UDP 例程

1.6.1 UDP 服务端

以下是如何在 RT-Thread 上使用 BSD socket 接口的一个 UDP 服务端例子，当把这个代码加入到 RT-Thread 时，它会自动向 finsh 命令行添加一个 udpserv 命令，在 finsh 上执行 udpserv()函数即可启动这个 UDP 服务端，它是在端口 5000 上进行监听。

当服务端接收到数据时，它将把数据打印到控制终端中；
如果服务端接收到 exit 字符串时，服务端将退出服务。

例子代码如下：

```
#include <rtthread.h>
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

void udpserv(void* paramemter)
{
   int sock;
   int bytes_read;
   char *recv_data;
   rt_uint32_t addr_len;
   struct sockaddr_in server_addr, client_addr;

   /* 分配接收用的数据缓冲 */
   recv_data = rt_malloc(1024);
   if (recv_data == RT_NULL)
   {
       /* 分配内存失败，返回 */
       rt_kprintf("No memory\n");
       return;
   }

   /* 创建一个socket，类型是SOCK_DGRAM，UDP类型 */
   if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
   {
       rt_kprintf("Socket error\n");

       /* 释放接收用的数据缓冲 */
       rt_free(recv_data);
       return;
   }
```

```
/* 初始化服务端地址 */
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(5000);
server_addr.sin_addr.s_addr = INADDR_ANY;
rt_memset(&(server_addr.sin_zero),0, sizeof(server_addr.sin_zero));

/* 绑定socket到服务端地址 */
if (bind(sock,(struct sockaddr *)&server_addr,
        sizeof(struct sockaddr)) == -1)
{
    /* 绑定地址失败 */
    rt_kprintf("Bind error\n");

    /* 释放接收用的数据缓冲 */
    rt_free(recv_data);
    return;
}

addr_len = sizeof(struct sockaddr);
rt_kprintf("UDPServer Waiting for client on port 5000...\n");

while (1)
{
    /* 从sock中收取最大1024字节数据 */
    bytes_read = recvfrom(sock, recv_data, 1024, 0,
                          (struct sockaddr *)&client_addr, &addr_len);
    /* UDP不同于TCP，它基本不会出现收取的数据失败的情况，除非设置了超时等待 */

    recv_data[bytes_read] = '\0'; /* 把末端清零 */

    /* 输出接收的数据 */
    rt_kprintf("\n(%s , %d) said : ",inet_ntoa(client_addr.sin_addr),
            ntohs(client_addr.sin_port));
    rt_kprintf("%s", recv_data);

    /* 如果接收数据是exit，退出 */
    if (strcmp(recv_data, "exit") == 0)
    {
        lwip_close(sock);

        /* 释放接收用的数据缓冲 */
        rt_free(recv_data);
        break;
    }
```

```
    }

    return;
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出udpserv函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(udpserv, startup udp server);
#endif
```

1.6.2 UDP 客户端

以下是如何在 RT-Thread 上使用 BSD socket 接口的一个 UDP 客户端例子。当把这个代码加入到 RT-Thread 时，它会自动向 finsh 命令行添加一个 udpclient 命令，在 finsh 上执行 udpclient（url，port）函数即可启动这个 TCP 服务端，url 指定了这个客户端连接到的服务端地址或域名，port 是相应的端口号。

当 UDP 客户端启动后，它将连续发送 5 次“This is UDP Client from RT-Thread.”的字符串给服务端，然后退出。

例子代码如下：

```
#include <rtthread.h>
#include <lwip/netdb.h> /* 为了解析主机名，需要包含netdb.h头文件 */
#include <lwip/sockets.h> /* 使用BSD socket，需要包含sockets.h头文件 */

const char send_data[] = "This is UDP Client from RT-Thread.\n"; /* 发送用到的数据 */
void udpclient(const char* url, int port, int count)
{
    int sock;
    struct hostent *host;
    struct sockaddr_in server_addr;

    /* 通过函数入口参数url获得host地址（如果是域名，会做域名解析） */
    host= (struct hostent *) gethostbyname(url);

    /* 创建一个socket，类型是SOCK_DGRAM，UDP类型 */
    if ((sock = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    {
        rt_kprintf("Socket error\n");
        return;
```

```
    }

    /* 初始化预连接的服务端地址 */
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(port);
    server_addr.sin_addr = *((struct in_addr *)host->h_addr);
    rt_memset(&(server_addr.sin_zero), 0,
sizeof(server_addr.sin_zero));

    /* 总计发送count次数据 */
    while (count)
    {
        /* 发送数据到服务远端 */
        sendto(sock, send_data, strlen(send_data), 0,
               (struct sockaddr *)&server_addr, sizeof(struct sockaddr));

        /* 线程休眠一段时间 */
        rt_thread_delay(50);

        /* 计数值减一 */
        count --;
    }

    /* 关闭这个socket */
    lwip_close(sock);
}

#ifdef RT_USING_FINSH
#include <finsh.h>
/* 输出udpclient函数到finsh shell中 */
FINSH_FUNCTION_EXPORT(udpclient, startup udp client);
#endif
```