# 网络开发应用笔记

## RT-THREAD 文档中心




WWW.RT-THREAD.ORG

Friday 28th September, 2018

# 目录

!!! abstract “摘要” 本应用笔记描述了如何在 RT-Thread 中利用标准化 API 来开发网络应用。

# 1 本文的目的和结构

## 1.1 本文的目的和背景

越来越多的单片机需要接入以太网来收发数据，市面上也有非常多的接入方案，可以用单片机加自带硬件协议栈的 PHY 芯片来接入网络，也可以单片机跑软件协议栈加 PHY 芯片来接入网络，不同的接入方案需要调用不同的 API，降低了上层应用的可移植性。

为了方便用户开发网络应用，RT-Thread 中引入了网络框架。并提供标准化 API 接口用于开发网络应用，同时，RT-Thread 还提供了数量丰富的网络组件包，方便用户快速开发自己的应用。

## 1.2 本文的结构

本文首先介绍了 RT-Thread 网络框架和标准化 API，然后介绍利用这些 API 实现的基础应用：tcp client 和 udp client，最后介绍了 RT-Thread 提供的网络组件包，并给出了在正点原子 STM32F4 探索者开发板上运行 NTP（通过网络获取时间）和 MQTT（通过 MQTT 收发数据）的代码示例。

# 2 网络框架介绍

RT-Thread 提供了一套网络管理框架，如**网络框架图**所示：

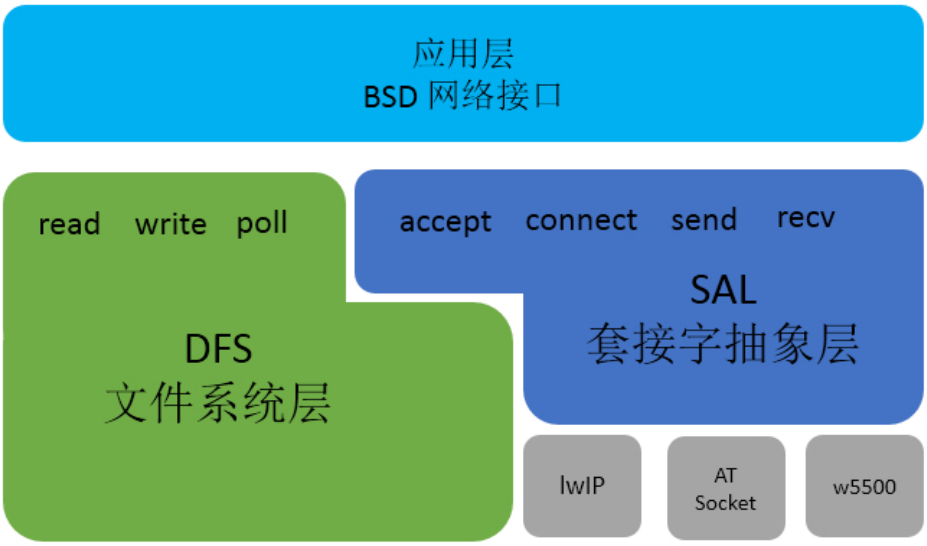


图 1: *sal_frame*

最顶层是网络应用层，提供一套标准 BSD Socket API ，如 socket、connect 等函数，用于系统中大部分网络开发应用。

第二部分为文件系统层，在 RT-Thread 系统中，通过 DFS 文件系统程序可以使用标准的接口函数实现不同的文件系统操作。网络套接字接口也是支持文件系统结构，使用网络套接字接口时创建的网络套接字描述符由文件系统统一管理，所以网络套接字描述符也可使用标准文件操作接口，文件系统层为上层应用层提供的接口有：read、write、close、poll/select 等。

第三部分为套接字抽象层，通过它 RT-Thread 系统能够适配下层不同的网络协议栈，并提供给上层统一的网络编程接口，方便不同协议栈的接入。套接字抽象层为上层应用层提供接口有：accept、connect、send、recv 等。

第四部分为协议栈层，该层包括几种常用的 TCP/IP 协议栈，例如嵌入式开发中常用的轻型 TCP/IP 协议栈 lwIP 以及 RT-Thread 自主研发的 AT Socket 网络功能实现等。这些协议栈或网络功能实现直接和硬件接触，完成数据从网络层到传输层的转化。

RT-Thread 的网络应用层提供的接口主要以标准 BSD Socket API 为主，这样能确保程序可以在 PC 上编写、调试，然后再移植到 RT-Thread 操作系统上。

# 3 API 介绍

## 3.1 BSD Socket API

BSD Socket（伯克利套接字）最初是由加州伯克利大学为 Unix 系统开发出来的。现代的操作系统基本上都实现了伯克利套接字接口，主流的编程语言也都支持利用 BSD Socket 开发网络应用。BSD Socket 可以说已经是连接互联网的标准接口了。

RT-Thread 也实现了 BSD Socket 接口 API，在别的操作系统或者编程语言中利用 BSD Socket 实现的网络应用，可以直接到 RT-Thread 中运行而不需要做任何修改。

## 3.2 创建套接字（socket）

```
int socket(int domain, int type, int protocol);
```

用于根据指定的地址族、数据类型和协议来分配一个套接字描述符及其所用的资源。

| 参数 | 描述 |
| --- | --- |
| domain | 协议族类型 |
| type | 协议类型 |
| protocol | 实际使用的运输层协议 |
| **返回** | **描述** |
| 0 | 成功，返回一个代表套接字描述符的整数 |
| -1 | 失败 |

**domain**

协议族

- PF_INET：IPv4
- PF_INET6：IPv6.

**type**

协议类型

- SOCK_STREAM：可靠的面向连接的服务或者 Stream Sockets
- SOCK_DGRAM：数据包服务或者 Datagram Sockets
- SOCK_RAW：网络层的原始协议

## 3.3 绑定套接字（bind）

```
int bind(int s, const struct sockaddr *name, socklen_t namelen);
```

用于将端口号和 IP 地址绑定带指定套接字上。当使用 socket() 创造一个套接字时，只是给定了协议族，并没有分配地址，在套接字接收来自其他主机的连接前，必须用 bind() 给它绑定一个地址和端口号。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| name | 指向 sockaddr 结构体的指针，代表要绑定的地址 |
| namelen | sockaddr 结构体的长度 |
| **返回** | **描述** |
| 0 | 成功 |
| -1 | 失败 |

## 3.4 监听套接字（listen）

```
int listen(int s, int backlog);
```

用于 TCP 服务器监听指定套接字连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| backlog | 表示一次能够等待的最大连接数目 |
| **返回** | **描述** |
| 0 | 成功 |
| -1 | 失败 |

## 3.5 接收连接（accept）

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

当应用程序监听来自其他主机的连接时，使用 accept() 函数初始化连接，accept() 为每个连接创立新的套接字并从监听队列中移除这个连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| addr | 客户端设备地址信息 |
| addrlen | 客户端设备地址结构体的长度 |

| 参数 | 描述 |
| --- | --- |
| **返回** | **描述** |
| 0 | 成功，返回新创建的套接字描述符 |
| -1 | 失败 |

## 3.6 建立连接（connect）

```
int connect(int s, const struct sockaddr *name, socklen_t namelen);
```

用于建立与指定 socket 的连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| name | 服务器地址信息 |
| namelen | 服务器地址结构体的长度 |
| **返回** | **描述** |
| 0 | 成功，返回新创建的套接字描述符 |
| -1 | 失败 |

## 3.7 TCP 数据发送（send）

```
int send(int s, const void *dataptr, size_t size, int flags);
```

发送数据，常用于 TCP 连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| dataptr | 发送的数据指针 |
| size | 发送的数据长度 |
| flags | 标志，一般为 0 |
| **返回** | **描述** |
| >0 | 成功，返回发送的数据的长度 |
| <=0 | 失败 |

## 3.8 TCP 数据接收（recv）

```
int recv(int s, void *mem, size_t len, int flags);
```

接收数据，常用于 TCP 连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| mem | 接收的数据指针 |
| len | 接收的数据长度 |
| flags | 标志，一般为 0 |
| **返回** | **描述** |
| >0 | 成功，返回接收的数据的长度 |
| =0 | 目标地址已传输完并关闭连接 |
| <0 | 失败 |

## 3.9 UDP 数据发送（sendto）

```
int sendto(int s, const void *dataptr, size_t size, int flags, const struct sockaddr *to, socklen_t tolen);
```

发送数据，常用于 UDP 连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| dataptr | 发送的数据指针 |
| size | 发送的数据长度 |
| flags | 标志，一般为 0 |
| to | 目标地址结构体指针 |
| tolen | 目标地址结构体长度 |
| **返回** | **描述** |
| >0 | 成功，返回发送的数据的长度 |
| <=0 | 失败 |

## 3.10 UDP 数据接收（recvfrom）

```
int recvfrom(int s, void *mem, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
```

接收数据，常用于 UDP 连接。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| mem | 接收的数据指针 |
| len | 接收的数据长度 |
| flags | 标志，一般为 0 |
| from | 接收地址结构体指针 |
| fromlen | 接收地址结构体长度 |
| **返回** | **描述** |
| >0 | 成功，返回接收的数据的长度 |
| =0 | 接收地址已传输完并关闭连接 |
| <0 | 失败 |

## 3.11 关闭套接字（closesocket）

```
int closesocket(int s);
```

关闭连接，释放资源。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| **返回** | **描述** |
| 0 | 成功 |
| -1 | 失败 |

## 3.12 按设置关闭套接字（shutdown）

```
int shutdown(int s, int how);
```

提供更多的权限控制套接字的关闭过程。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| how | 套接字控制的方式 |
| **返回** | **描述** |
| 0 | 成功 |
| -1 | 失败 |

**how**

- 0：停止接收当前数据，并拒绝以后的数据接收；
- 1：停止发送数据，并丢弃未发送的数据；
- 2：停止接收和发送数据。

## 3.13 设置套接字选项（setsockopt）

```
int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

设置套接字模式，修改套接字配置选项。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| level | 协议栈配置选项 |
| optname | 需要设置的选项名 |
| optval | 设置选项值的缓冲区地址 |
| optlen | 设置选项值的缓冲区长度 |
| **返回** | **描述** |
| =0 | 成功 |
| <0 | 失败 |

**level**

- SOL_SOCKET：套接字层
- IPPROTO_TCP：TCP 层
- IPPROTO_IP：IP 层

**optname**

- SO_KEEPALIVE：设置保持连接选项
- SO_RCVTIMEO：设置套接字数据接收超时
- SO_SNDTIMEO：设置套接数据发送超时

## 3.14 获取套接字选项（getsockopt）

```
int getsockopt(int s, int level, int optname, void *optval, socklen_t *
optlen);
```

获取套接字配置选项。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| level | 协议栈配置选项 |
| optname | 需要设置的选项名 |
| optval | 获取选项值的缓冲区地址 |
| optlen | 获取选项值的缓冲区长度地址 |
| **返回** | **描述** |
| =0 | 成功 |
| <0 | 失败 |

## 3.15 获取远端地址信息（getpeername）

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

获取与套接字相连的远端地址信息。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| name | 接收信息的地址结构体指针 |
| namelen | 接收信息的地址结构体长度 |
| **返回** | **描述** |
| =0 | 成功 |
| <0 | 失败 |

## 3.16 获取本地地址信息（getsockname）

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

获取本地套接字地址信息。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| name | 接收信息的地址结构体指针 |
| namelen | 接收信息的地址结构体长度 |
| **返回** | **描述** |
| =0 | 成功 |
| <0 | 失败 |

## 3.17 配置套接字参数（ioctlsocket）

```
int ioctlsocket(int s, long cmd, void *arg);
```

设置套接字控制模式。

| 参数 | 描述 |
| --- | --- |
| s | 套接字描述符 |
| cmd | 套接字操作命令 |
| arg | 操作命令所带参数 |
| **返回** | **描述** |
| =0 | 成功 |
| <0 | 失败 |

**cmd**

- FIONBIO：开启或关闭套接字的非阻塞模式，arg 参数 1 为开启非阻塞，0 为关闭非阻塞。

## 3.18 调试 API

这里介绍下 RT-Thread 提供的三个网络信息查看命令，在 shell 中输入命令即可很方便的查看网络连接状况，方便用户进行调试。

### 3.18.1. ifconfig

ifconfig可以打印出板子现在的网络连接状态，IP 地址，网关地址，dns 等信息。

```
msh />ifconfig
network interface: e0 (Default)
MTU: 1500
MAC: 00 04 9f 05 44 e5
FLAGS: UP LINK_UP ETHARP
ip address: 192.168.12.127
gw address: 192.168.10.1
net mask  : 255.255.0.0
dns server #0: 192.168.10.1
dns server #1: 223.5.5.5
msh />
```

图 2: *an010_ifconfig*

### 3.18.2. netstate

netstate可以打印出板子所有的 TCP / IP 连接信息

```
Active PCB states:
#0 192.168.12.186:49153 <==> 139.196.135.135:1883 snd_nxt 0x00001
AEA rcv_nxt 0x9D4D8F35 state: ESTABLISHED
Listen PCB states:
TIME-WAIT PCB states:
Active UDP PCB states:
#0 4 0.0.0.0:68 <==> 0.0.0.0:67
```

图 3: *an011_netstate*

### 3.18.3. dns

dns可以打印出现在使用的 dns 服务器地址，也输入 dns 服务器 IP 地址来手动设置 dns 服务器地址

```
msh />dns                                  查看dns
dns server #0: 192.168.10.1
dns server #1: 223.5.5.5
msh />dns 114.114.114.114                  设置dns
dns : 114.114.114.114
msh />dns                                  查看dns
dns server #0: 114.114.114.114
dns server #1: 223.5.5.5
msh />
```

图 4: *an011_dns*

# 4 准备工作

- 准备RT-Thread 源码
- 准备ENV
- 一块能上网的开发板, 这里以正点原子 STM32F4 探索者开发板为例
- 移植好网络底层驱动，驱动移植可以参考网络协议栈驱动移植笔记
- 网络调试工具

## 4.1 硬件连接准备

RT-Thread 的 BSP 默认启用了 DHCP 功能，需要有 DHCP 服务器来分配 IP 地址，常见的连接拓展如图：

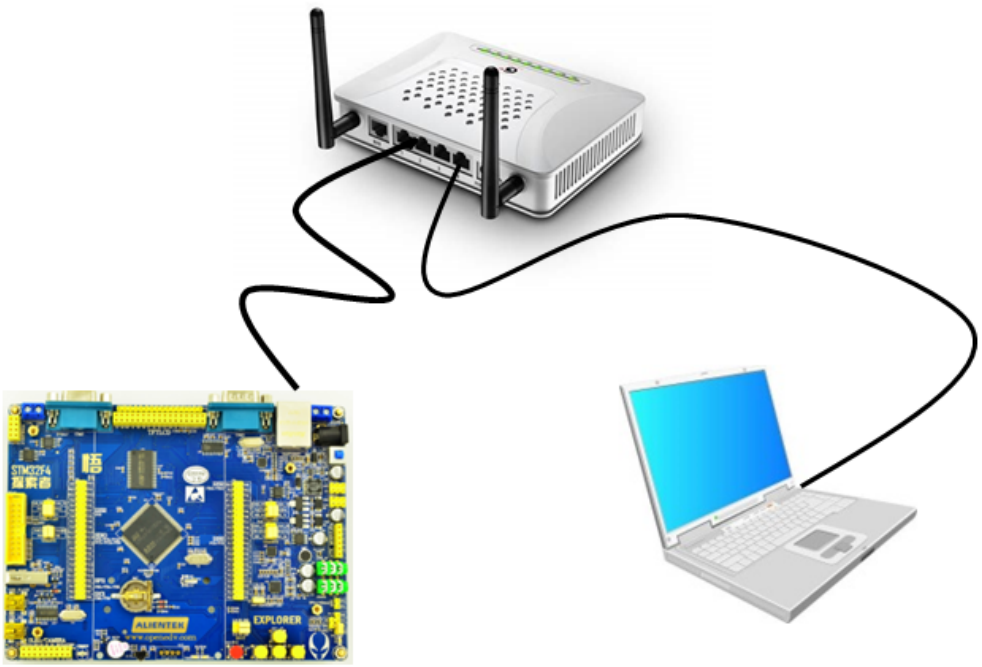

图 5: *eth_RJ45*

如果没有方便的实际环境，也可以先通过 ENV 配置固定 IP，然后用网线直接连接到调试用的电脑。

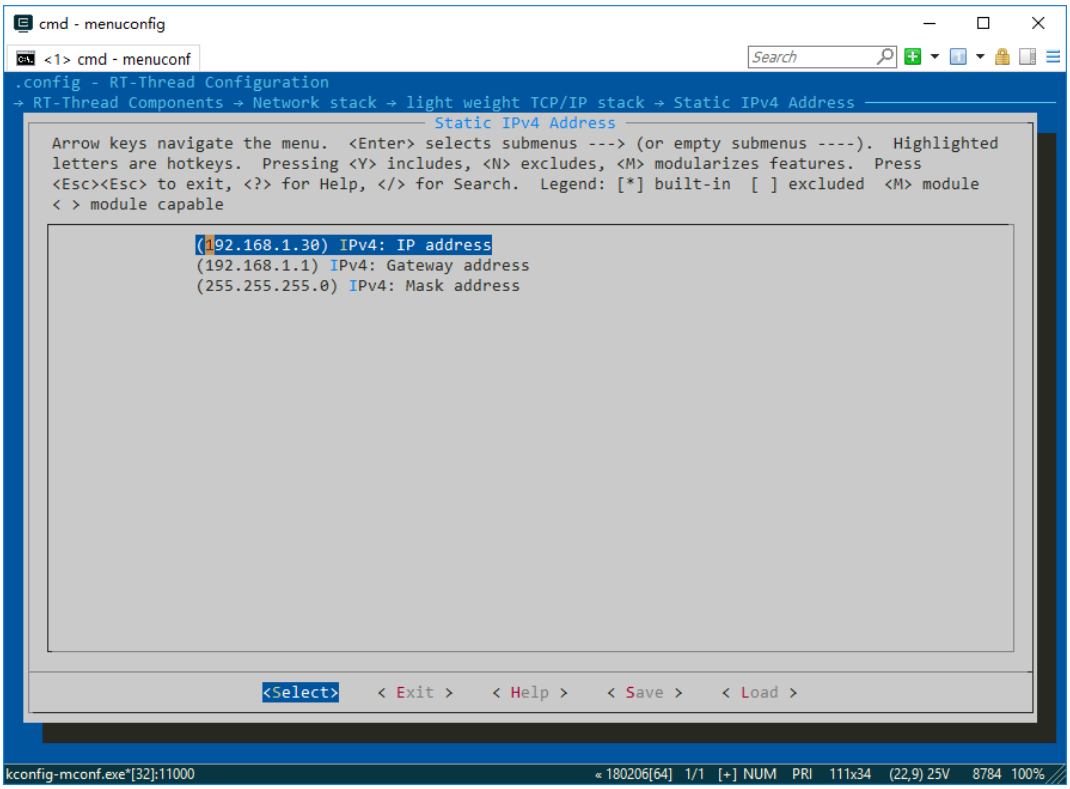


图 6: *ipadress*

电脑和开发板需要设置同网段的 IP 地址。

## 4.2 ENV 配置

RT-Thread 可以很方便的通过 ENV 来配置和生成工程

- 打开 env，进入 `rt-thread/bsp/stm32f40x` 目录
- 在 env 命令行中输入 `set RTT_CC=keil` ，设置工具链类型为 keil
- 在 env 命令行中输入 `menuconfig`，进入配置界面
- 在 RT-Thread Kernel -> Kernel Device object 页面修改控制台输出为自己板子引出的串口号

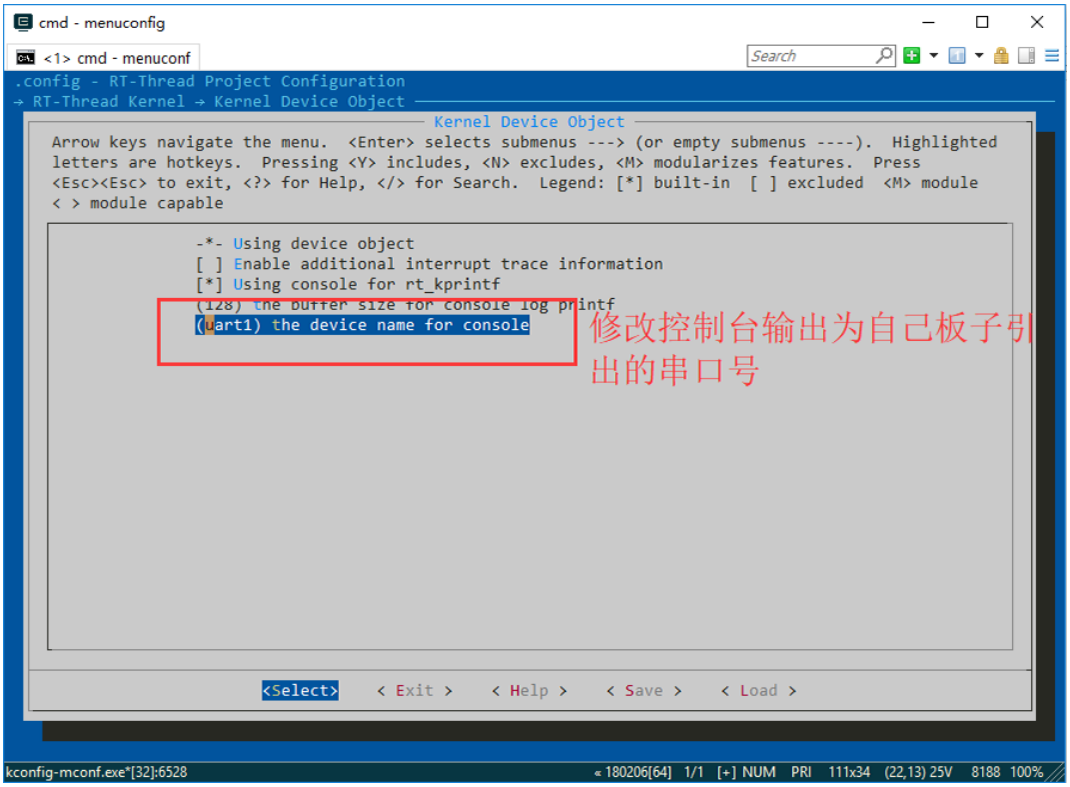


图 7: *ENV_uart*

- 在 RT-Thread Components -> Device virtual file system 页面查看最大文件打开数量是否小于 16，如果小于 16，则需要将值改大。

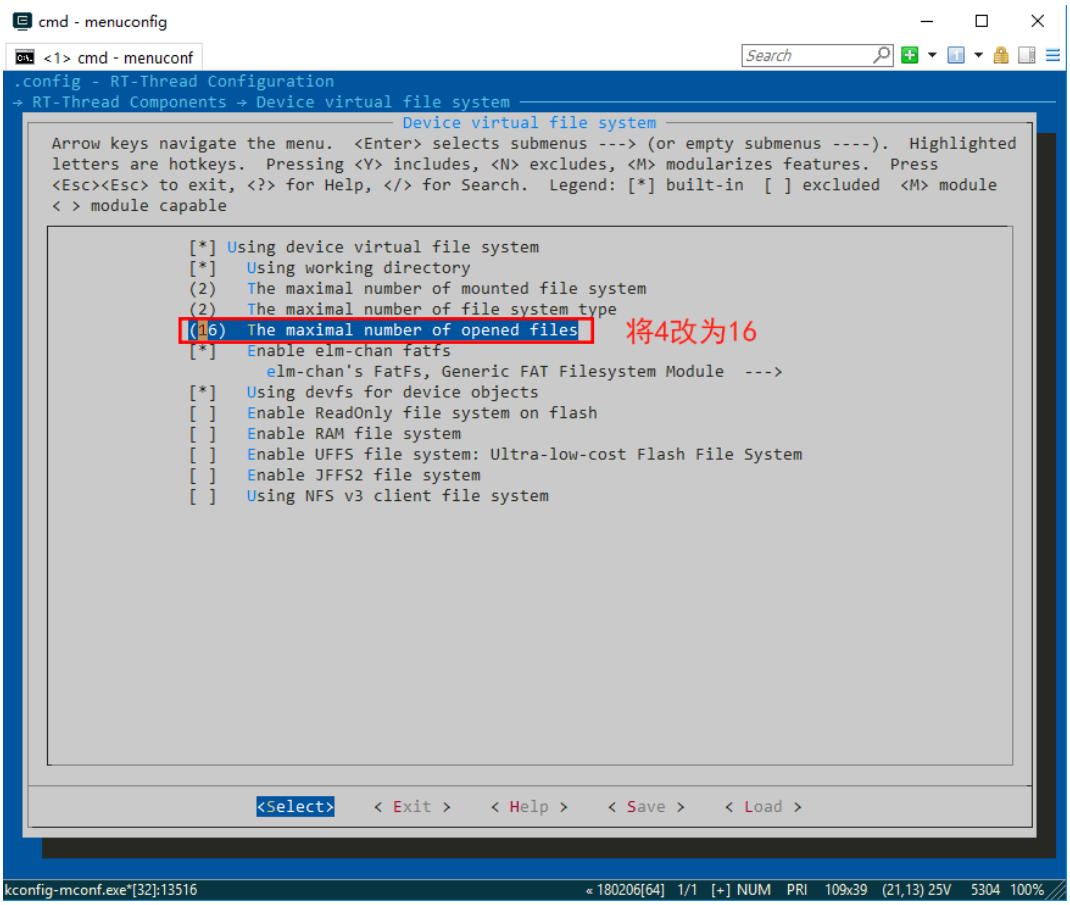


图 8: *menuconfig_filesystem*

在 RT-Thread Components -> Network -> Socket abstraction layer 页面启用 sal 层并使能 BSD socket

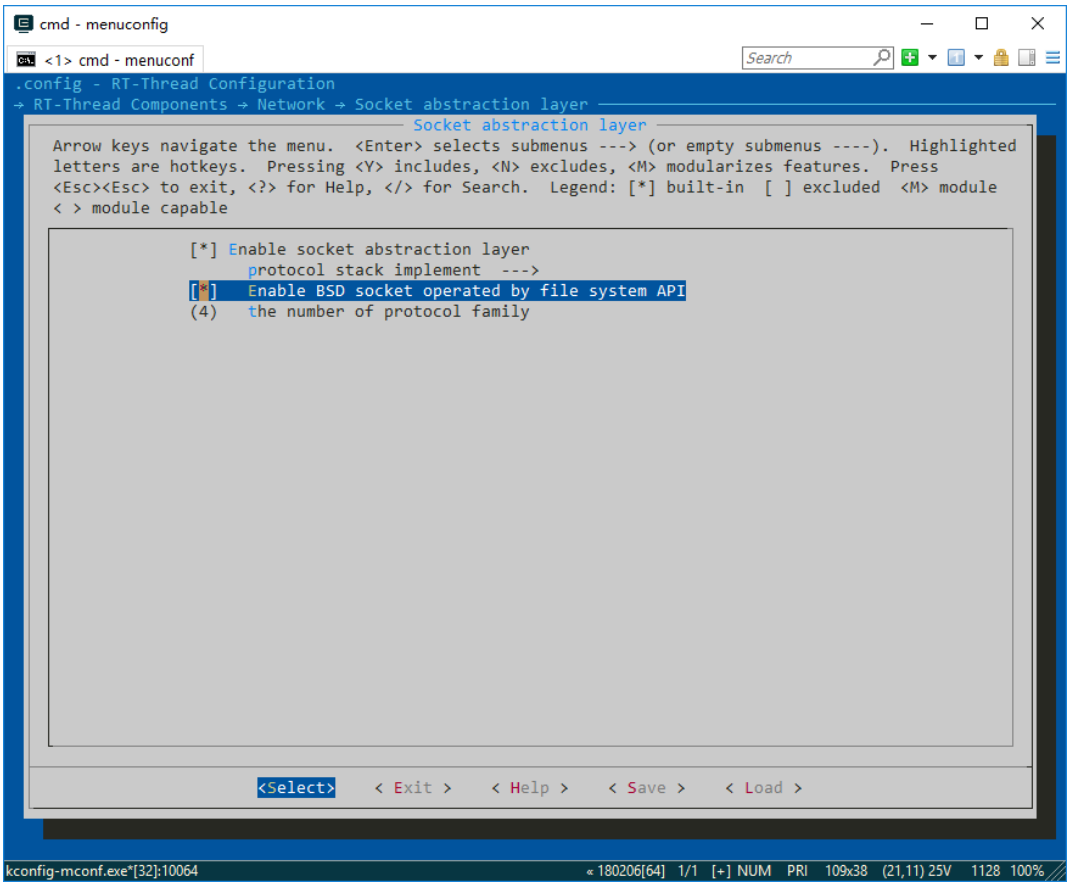


图 9: *an011_sai*

- 在 RT-Thread Components -> Network -> light weight TCP/IP stack 页面启用 lwip

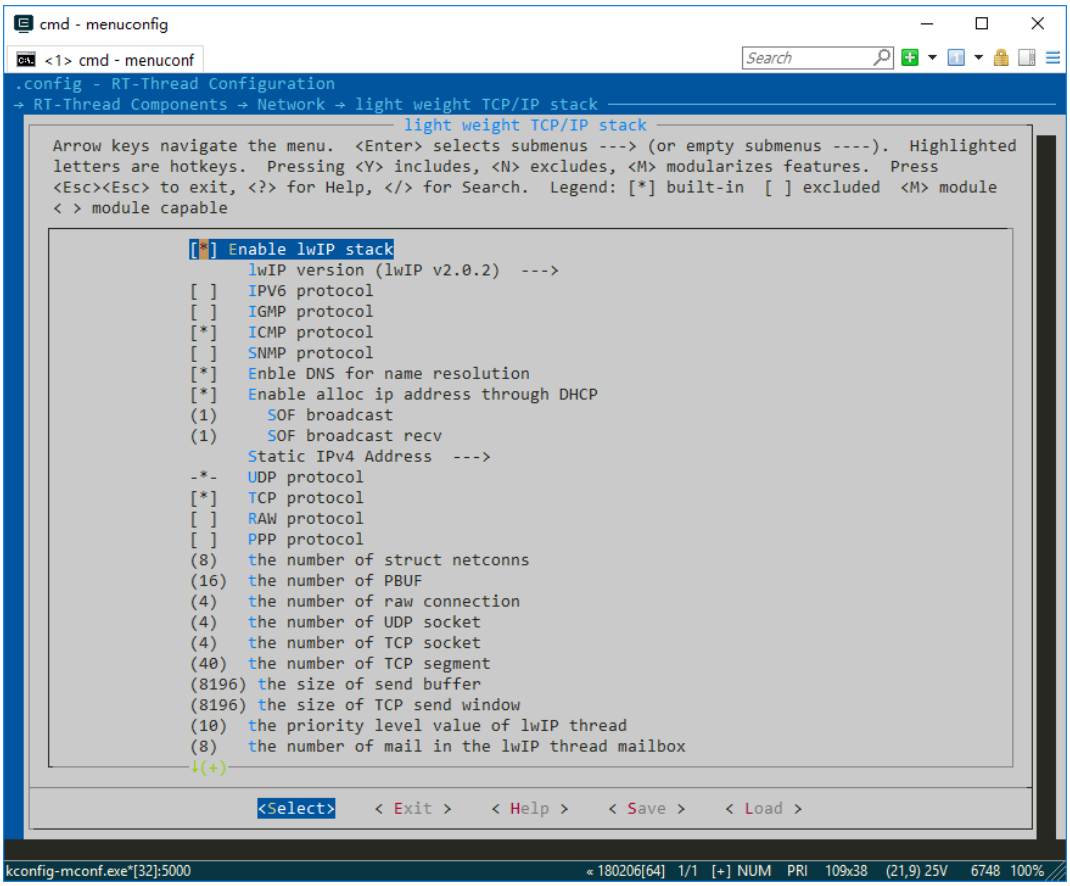


图 10: *an011_lwip*

- 在 RT-Thread online packages -> misellaneous packages -> samples: RT-Thread kernel and components samples->network sample options 页面使能 tcp client 和 udp client（基础应用）

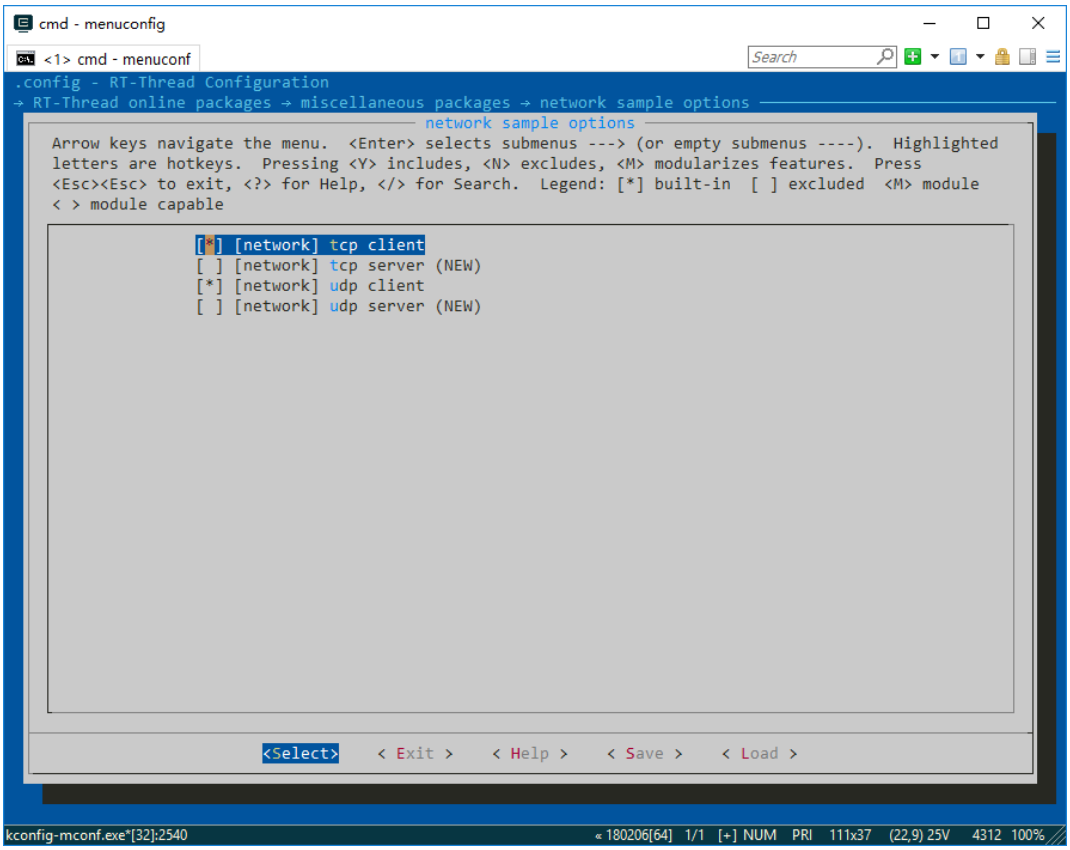


图 11: *ENV_client*

- 在 RT-Thread online packages -> IoT - internet of things -> netutils: Networking utilities for RT-Thread 页面使能 ntp（高级应用）

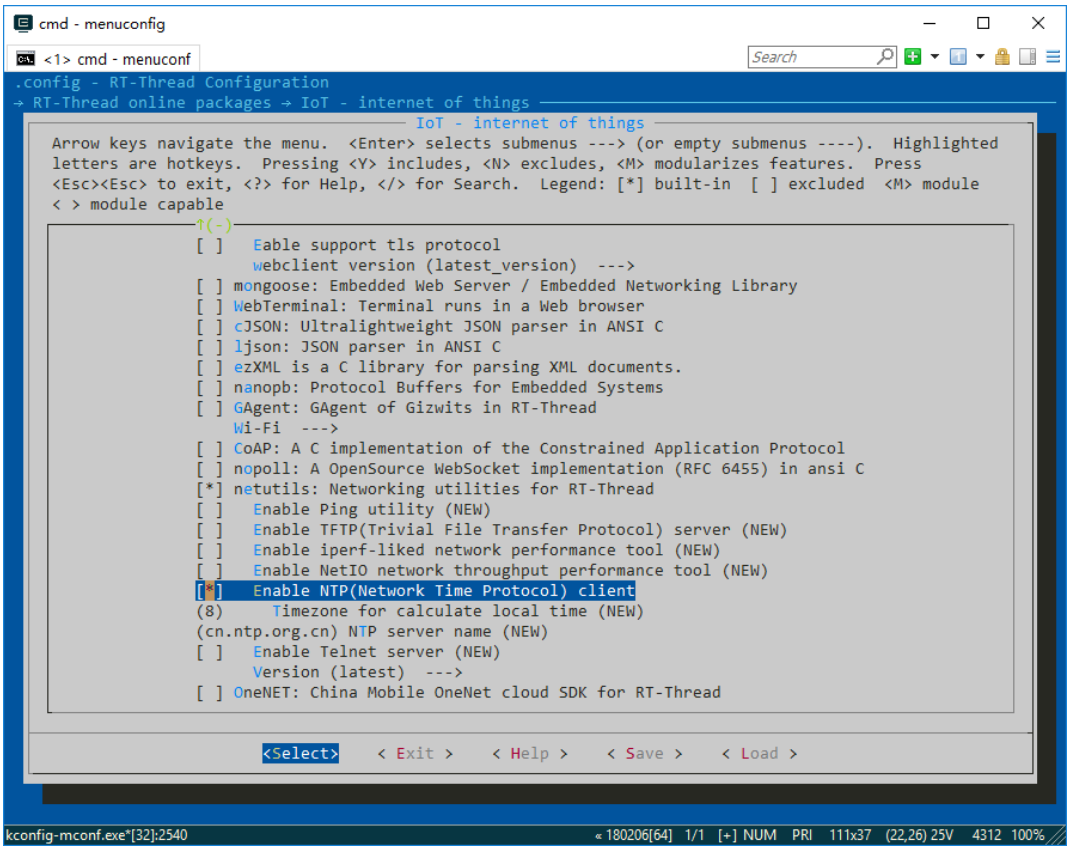


图 12: *ENV_ntp*

- 使能 mqtt（高级应用）

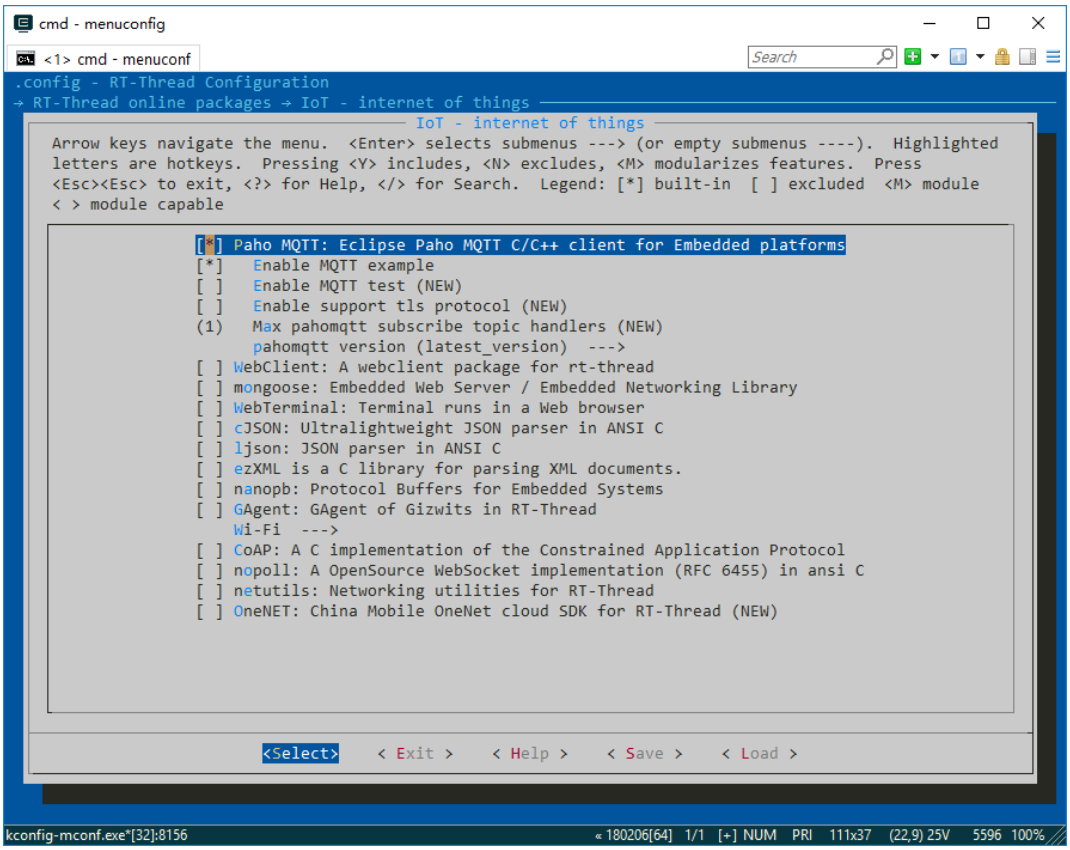


图 13: *ENV_mqtt*

- 按 ESC 退出配置界面
- 在 env 命令行中输入 scons –target=mdk5 -s 生成 mdk5 工程。
- 打开工程，编译
- 下载代码

## 4.3 网络测试

将 env 生成的工程编译后下载到板子上，可以看到网口的两盏灯会亮起，一盏会闪烁，说明 PHY 已经正常初始化了。

在 shell 中输入 ifconfig 可以打印板子的网络状态，正常获取到 ip 即表示网络驱动正常，准备工作完成。

```
msh />ifconfig
network interface: e0 (Default)
MTU: 1500
MAC: 00 04 9f 05 44 e5
FLAGS: UP LINK_UP ETHARP
ip address: 192.168.12.127
gw address: 192.168.10.1
net mask  : 255.255.0.0
dns server #0: 192.168.10.1
dns server #1: 223.5.5.5
msh />
```

图 14: *ifconfig*

# 5 基础应用

在实际应用中，单片机一般作为客户端去和服务器进行数据交换，在这里，以 tcp client 和 udp client 为例进行讲解。

## 5.1 tcpclient

这个例程展示了如何创建一个 TCP 客户端，跟远端服务器进行通信。

在 shell 中输入 tcpclient URL PORT 来连接服务器

程序功能：接收并显示从服务端发送过来的信息，接收到开头是 'q' 或 'Q' 的信息则退出程序

### 5.1.1. 源码解析

```
void tcpclient(int argc, char **argv)
{
    int ret;
    char *recv_data;
    struct hostent *host;
    int sock, bytes_received;
    struct sockaddr_in server_addr;
    const char *url;
    int port;

    /* 接收到的参数小于3个 */
    if (argc < 3)
    {
        rt_kprintf("Usage: tcpclient URL PORT\n");
```

```
        rt_kprintf("Like: tcpclient 192.168.12.44 5000\n");
        return ;
    }

    url = argv[1];
    port = strtoul(argv[2], 0, 10);

    /* 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(BUFSZ);
    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 fail!\n");
        closesocket(sock);

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

```
    }

    while (1)
    {
        /* 从sock连接中接收最大BUFSZ - 1字节数据 */
        bytes_received = recv(sock, recv_data, BUFSZ - 1, 0);
        if (bytes_received < 0)
        {
            /* 接收失败，关闭这个连接 */
            closesocket(sock);
            rt_kprintf("\nreceived error,close the socket.\r\n");

            /* 释放接收缓冲 */
            rt_free(recv_data);
            break;
        }
        else if (bytes_received == 0)
        {
            /* 打印recv函数返回值为0的警告信息 */
            rt_kprintf("\nReceived warning,recv function return 0.\r\n");

            continue;
        }

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

        if (strncmp(recv_data, "q", 1) == 0 || strncmp(recv_data, "Q", 1)
            == 0)
        {
            /* 如果是首字母是q或Q，关闭这个连接 */
            closesocket(sock);
            rt_kprintf("\n got a 'q' or 'Q',close the socket.\r\n");

            /* 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("\nReceived data = %s ", recv_data);
        }

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

```
        if (ret < 0)
        {
            /* 接收失败，关闭这个连接 */
            closesocket(sock);
            rt_kprintf("\nsend error,close the socket.\r\n");

            rt_free(recv_data);
            break;
        }
        else if (ret == 0)
        {
            /* 打印send函数返回值为0的警告信息 */
            rt_kprintf("\n Send warning,send function return 0.\r\n");
        }
    }
    return;
}
```

### 5.1.2. 运行结果

用网络调试工具在电脑上搭建一个 TCP 服务器，记录下打开的端口

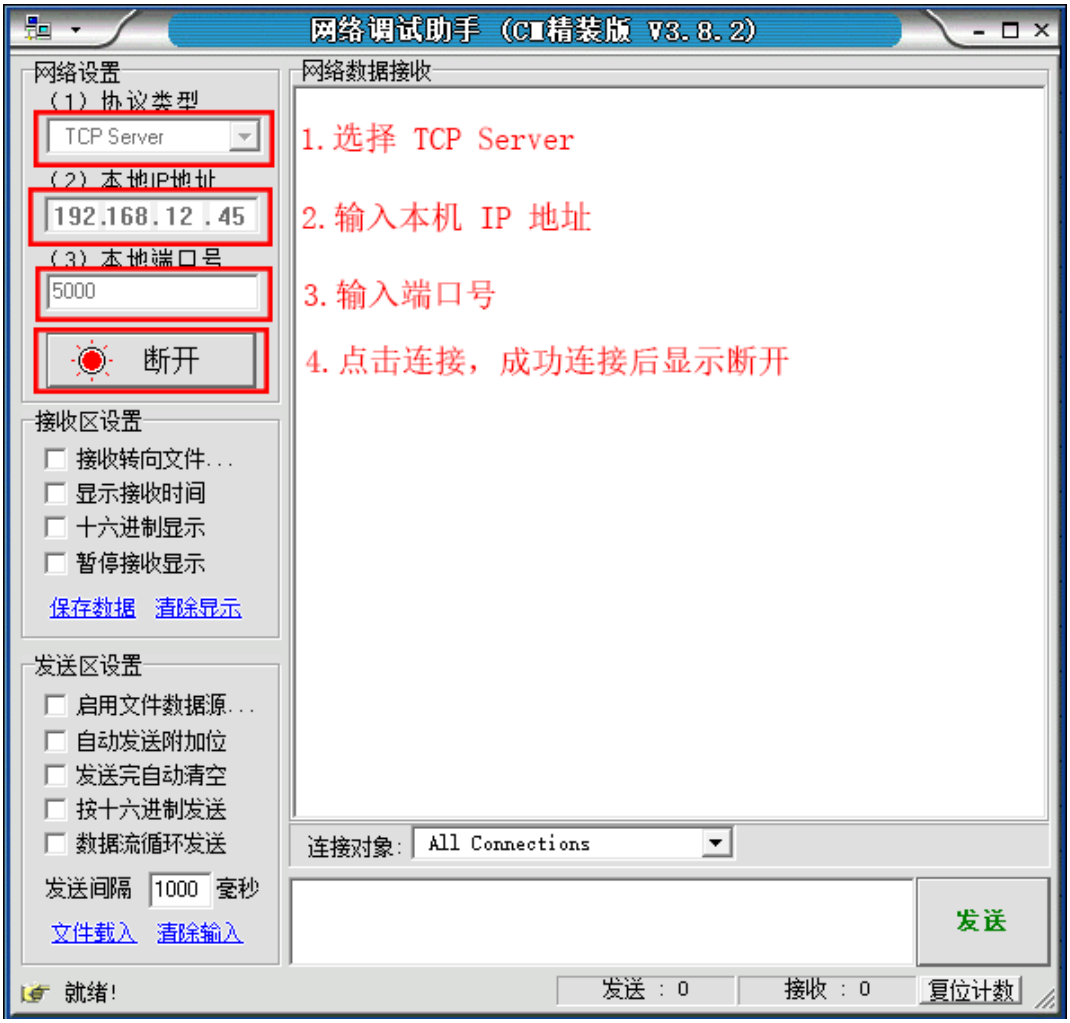


图 15: *TCP Server_set*

在 shell 中输入 tcpclient PC 的 IP 地址刚才记录下的端口号

```
msh />tcpclient 192.168.12.45 5000
```

利用服务器发送 Hello RT-Thread! ，shell 中会显示收到的信息

```
msh />tcpclient 192.168.12.45 5000

Received data = Hello RT-Thread!
```

图 16: *tcpclient_shell*

服务器会收到 This is TCP Client from RT-Thread. 的消息

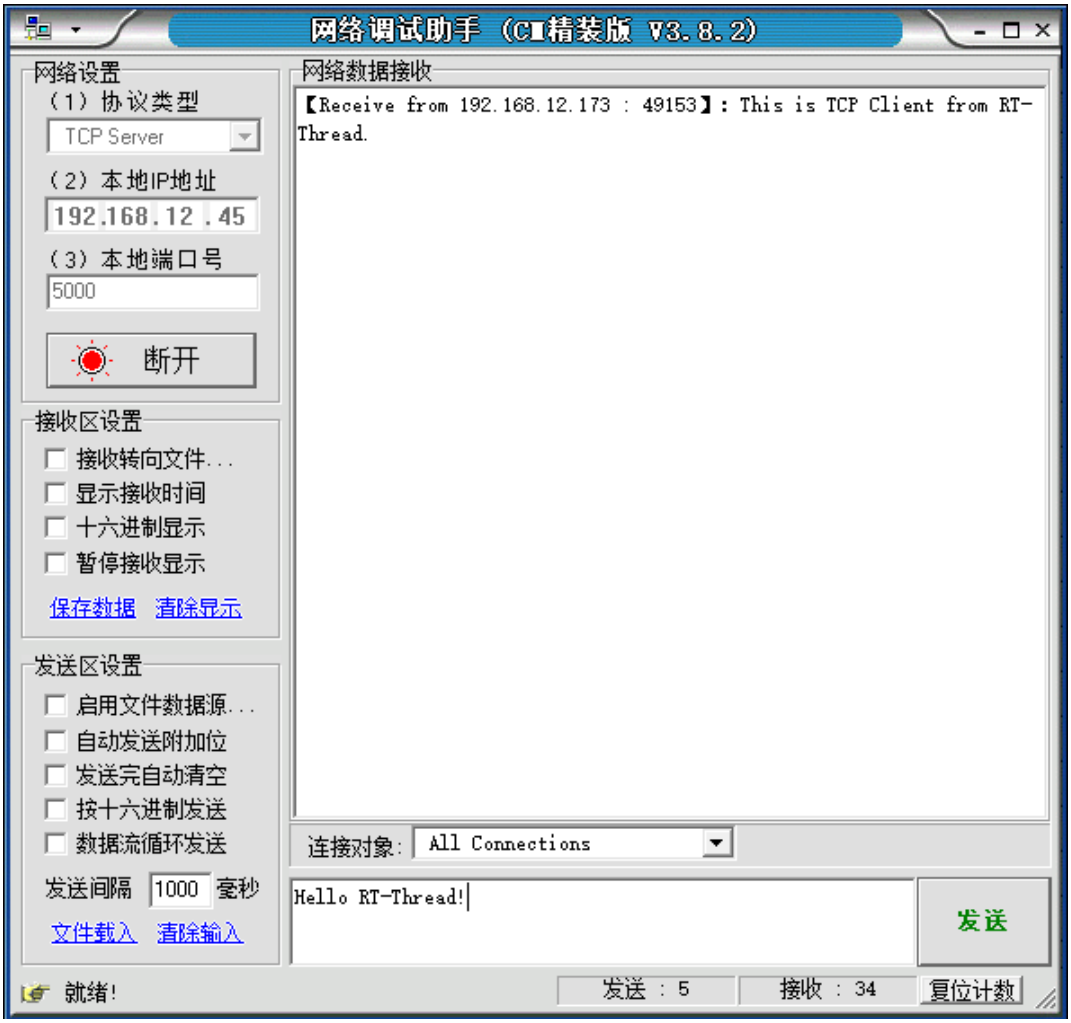


图 17: *tcpclient*

## 5.2 udpclient

这个例程展示了如何创建一个 UDP 客户端，给远端服务器发送数据。

在 shell 中输入 udpclient URL PORT 来连接服务器

程序功能：给服务端发送信息（默认 10 条）

### 5.2.1. 源码解析

```
void udpclient(int argc, char **argv)
{
    int sock, port, count;
    struct hostent *host;
    struct sockaddr_in server_addr;
```

```
    const char *url;

    /* 接收到的参数小于3个 */
    if (argc < 3)
    {
        rt_kprintf("Usage: udpclient URL PORT [COUNT = 10]\n");
        rt_kprintf("Like: tcpclient 192.168.12.44 5000\n");
        return ;
    }

    url = argv[1];
    port = strtoul(argv[2], 0, 10);

    if (argc > 3)
        count = strtoul(argv[3], 0, 10);
    else
        count = 10;

    /* 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 */
    closesocket(sock);
}
```

### 5.2.2. 运行结果

用网络调试工具在电脑上搭建一个 UDP 服务器，记录下打开的端口

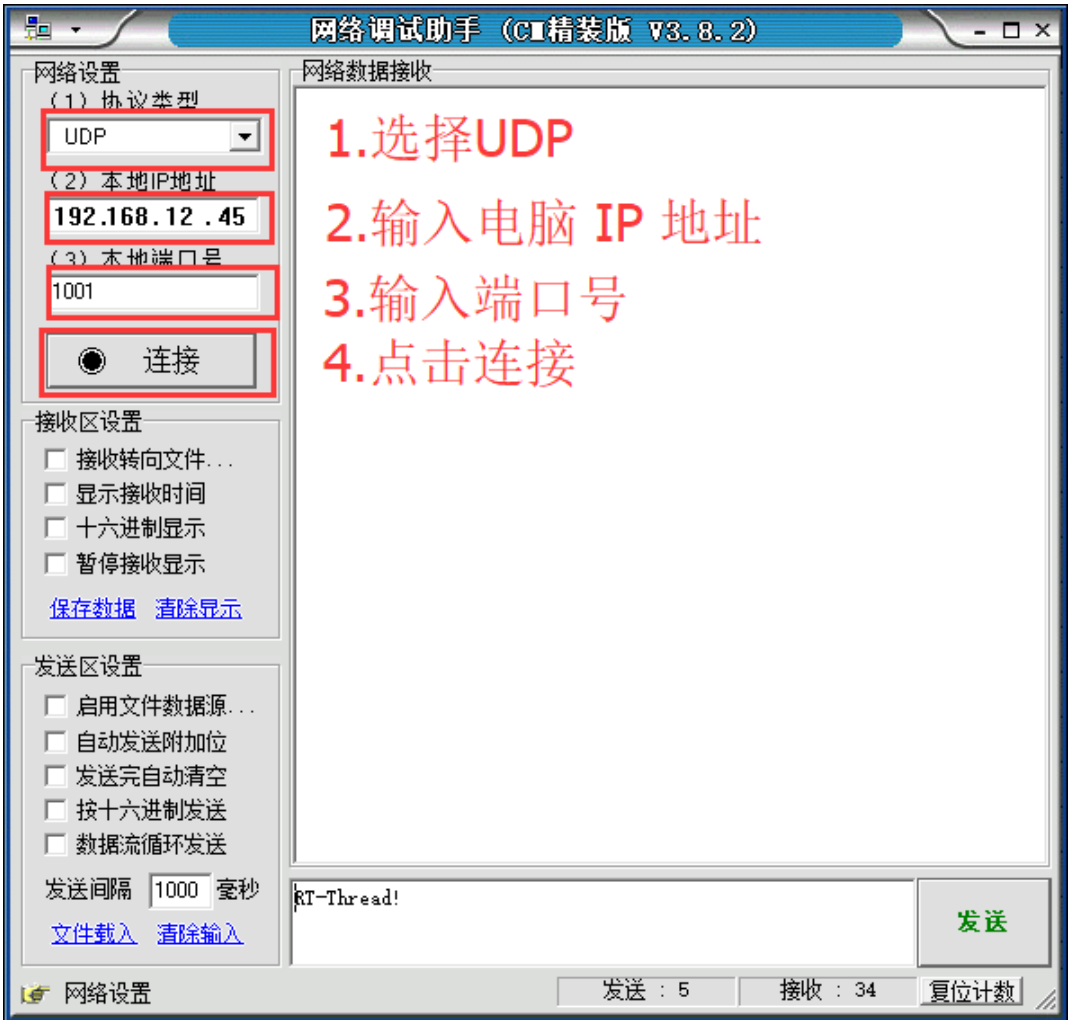


图 18: *udp_set*

在 shell 中输入 udpclient PC 的 IP 地址刚才记录下的端口号

```
udpclient 192.168.12.45 1001
```

服务器会收到 10 条 This is UDP Client from RT-Thread. 的消息

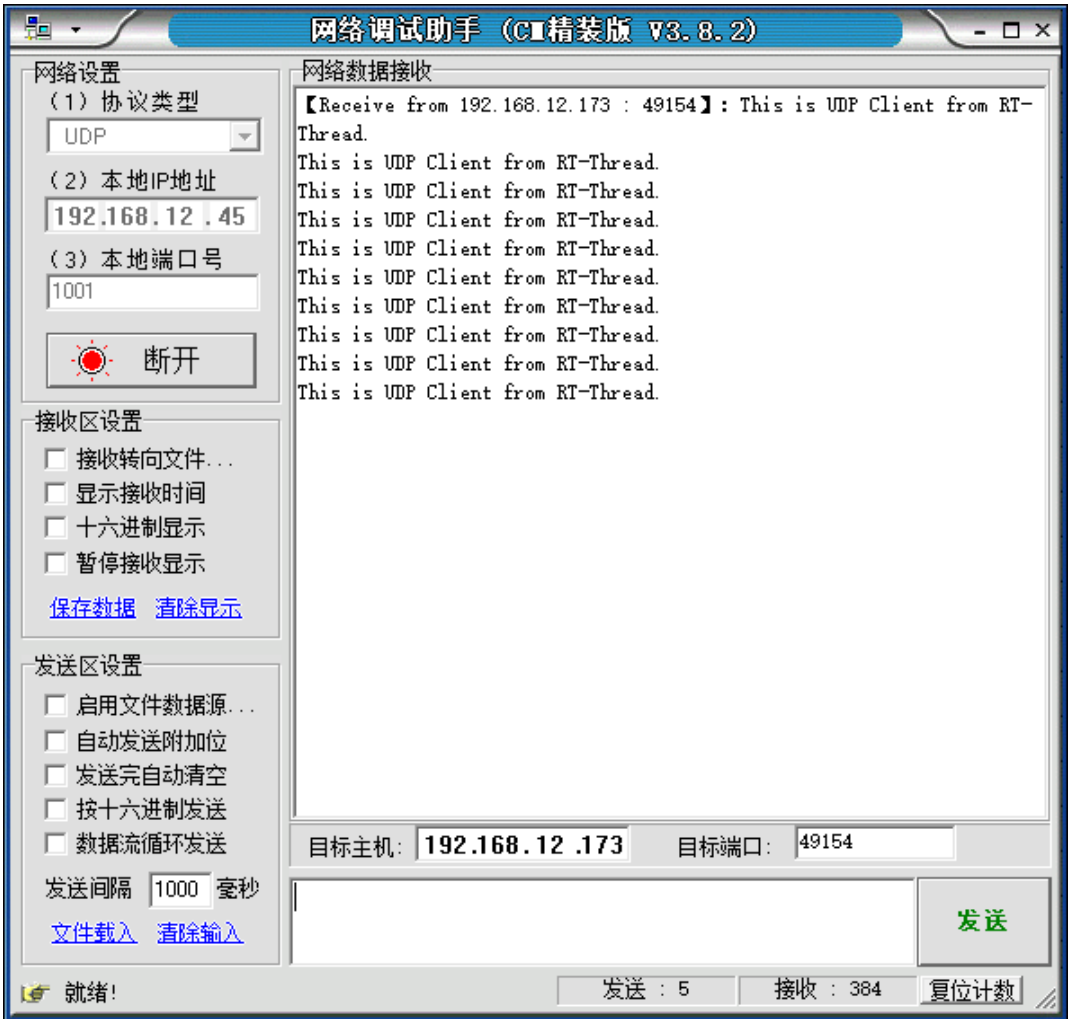


图 19: *udpclient*

# 6 高级应用

为了方便网络应用开发，RT-Thread 提供了丰富的网络组件包，例如：netutils 网络小工具集，webclient，cJSON，paho-mqtt等等，用户可以根据需求直接在 env 中使能即可使用各个组件包，省去了自己移植的过程，加速网络应用开发。

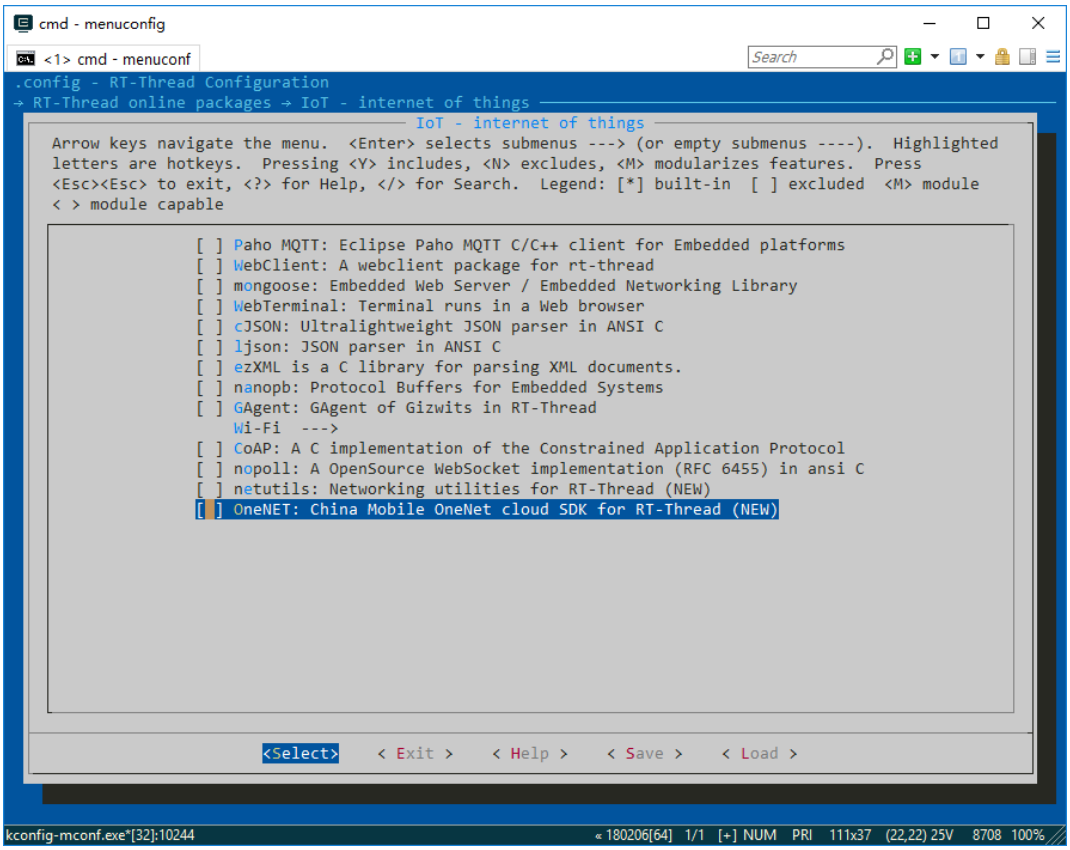


图 20: *ENV_IoT*

我们这里以 netutils 网络小工具集中的 NTP（时间同步）小工具和paho-mqtt为例进行讲解

## 6.1 NTP

NTP(Network Time Protocol) 是网络时间协议，它是用来同步网络中各个计算机时间的协议。

RT-Thread 实现了 NTP 客户端，可以通过网络获取本地时间，并同步板子的 RTC 时间。

ENV 的配置参考前面准备工作章节的 ENV 配置

在 msh 中输入 `ntp_sync` 即可从默认的 NTP 服务器 (cn.ntp.org.cn ) 获取本地时间，默认时区为东八时区

```
msh />ntp_sync
```

如果输入 `ntp_sync` 后提示超时或者连接失败，可以在 `ntp_sync` 后面输入 NTP 服务器地址，程序将从新的服务器获取时间。

```
msh />ntp_sync edu.ntp.org.cn
```

```
msh />ntp_sync
Get local time from NTP server: Wed Jun  6 16:06:01 2018
The system time is updated. Timezone is 8.
msh />ntp_sync edu.ntp.org.cn
Get local time from NTP server: Wed Jun  6 16:06:09 2018
The system time is updated. Timezone is 8.
msh />
```

图 21: *gettime*

## 6.2 MQTT

Paho MQTT 是 Eclipse 实现的 MQTT 协议的客户端，本软件包是在 Eclipse paho-mqtt 源码包的基础上设计的一套 MQTT 客户端程序。

MQTT 使用发布/订阅消息模式，发送消息时要指定发给哪个主题名（Topic Name），接收消息前要订阅一个主题名，然后才能接收到发送给这个主题名的消息内容。

RT-Thread MQTT 客户端功能特点：

- 断线自动重连
- pipe 模型，非阻塞 API
- 事件回调机制
- TLS 加密传输

ENV 的配置参考前面准备工作章节的 ENV 配置

在 msh 中输入 `mq_start` 命令, 客户端会自动连接服务器，并订阅`/mqtt/test`主题

```
msh />mq_start
```

通过 `mq_pub` 命令可以发送消息给所有订阅了`/mqtt/test`的客户端，我们利用 `mq_pub` 发送 RT-Thread!

```
msh />mq_pub  RT-Thread!
```

由于我们之前订阅了`/mqtt/test`主题，shell 很快会显示服务器发来的 RT-Thread! 消息。

```
msh />mq_start
[MQTT] inter mqtt_connect_callback!
[MQTT] ipv4 address port: 1883
[MQTT] HOST = 'iot.eclipse.org'
msh />[MQTT] MQTT server connect success
[MQTT] Subscribe #0 /mqtt/test OK!
[MQTT] inter mqtt_online_callback!
msh />mq_pub RT-Thread!
msh />[MQTT] mqtt sub callback: /mqtt/test RT-Thread!
```

图 22: *mqtttest*