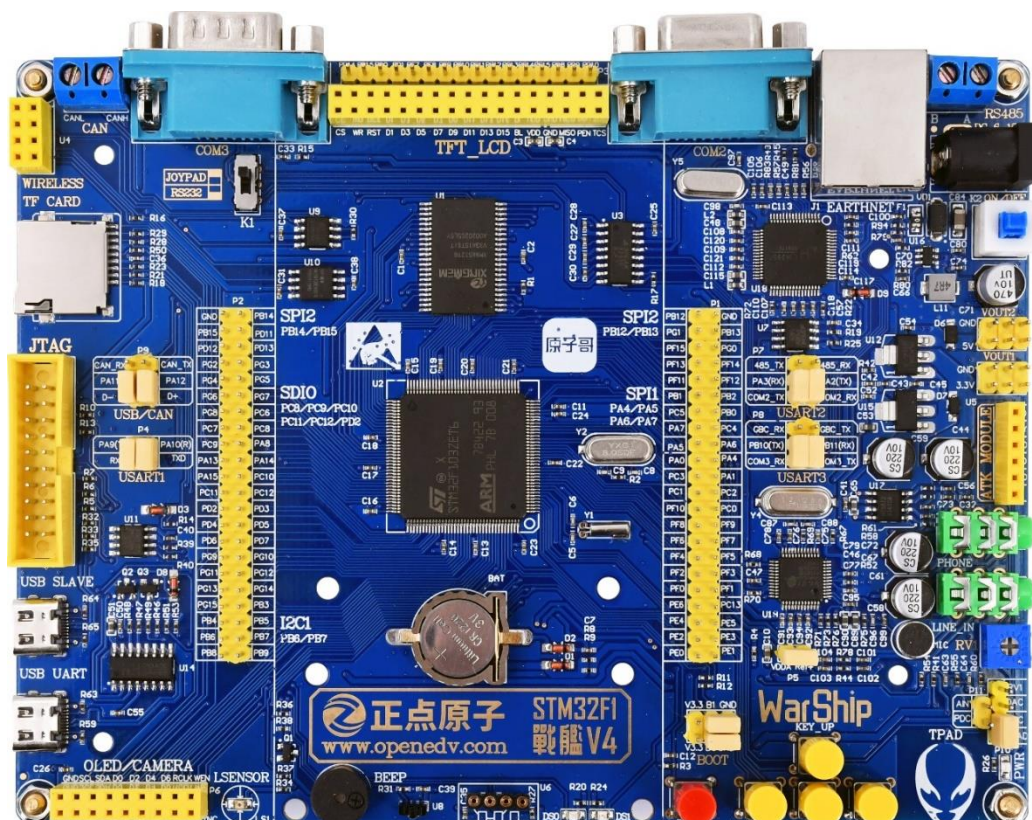


# 网络开发指南

## V1.0

-ALIENTEK 战舰 以太网开发教程



修订历史:

版本	日期	修改内容
V1.0	2022/10/10	第一次发布



正点原子公司名称：广州市星翼电子科技有限公司

原子哥在线教学平台：[www.yuanzige.com](http://www.yuanzige.com)

开源电子网 / 论坛：[www.openedv.com/forum.php](http://www.openedv.com/forum.php)

正点原子官方网站：[www.alientek.com](http://www.alientek.com)

正点原子淘宝店铺：<https://openedv.taobao.com>

正点原子 B 站视频：<https://space.bilibili.com/394620890>

电话：020-38271790 传真：020-36773971

请下载原子哥 APP，数千讲视频免费学习，更快更流畅。

请关注正点原子公众号，资料发布更新我们会通知。



扫码下载“原子哥”APP



扫码关注正点原子公众号

第一章 全硬件的 TCP/IP 协议栈简介.....	6
1.1 以太网接入单片机方案 .....	6
1.2 以太网接口芯片 CH395Q 简介 .....	7
1.3 以太网接口芯片 CH395Q 命令简介 .....	9
1.4 以太网接口芯片 CH395Q 寄存器配置与使用 .....	14
1.5 移植 CH395Q 源码 .....	36
第二章 UDP 实验 .....	41
2.1 UDP 配置流程 .....	41
2.2 UDP 实验 .....	41
2.2.1 硬件设计 .....	41
2.2.2 程序设计 .....	42
2.2.3 下载验证 .....	47
第三章 TCP_Client 实验 .....	49
3.1 TCPClient 配置流程 .....	49
3.2 TCPClient 实验 .....	49
3.2.1 硬件设计 .....	49
3.2.2 程序设计 .....	50
3.2.3 下载验证 .....	55
第四章 TCP_Server 实验 .....	57
4.1 TCPServer 配置流程 .....	57
4.2 TCPServer 实验 .....	57
4.2.1 硬件设计 .....	57
4.2.2 程序设计 .....	58
4.2.3 下载验证 .....	63
第五章 WebServer 实验.....	65
5.1 WebServer 简介 .....	65
5.2 WebServer 实验 .....	65

5.2.1 硬件设计 .....	65
5.2.2 软件设计 .....	65
5.2.3 下载验证 .....	71
<b>第六章 NTP 实时时间实验.....</b>	<b>72</b>
6.1 NTP 简介 .....	72
6.2 NTP 实验.....	74
6.2.1 硬件设计 .....	74
6.2.2 软件设计 .....	74
6.2.3 下载验证 .....	81
<b>第七章 基于 MQTT 协议连接 OneNET 服务器 .....</b>	<b>82</b>
7.1 MQTT 协议简介 .....	82
7.1.1 MQTT 协议实现原理 .....	82
7.1.2 移植 MQTT 协议 .....	84
7.2 配置 OneNET 平台 .....	85
7.3 工程配置 .....	89
7.4 基于 OneNET 平台 MQTT 实验 .....	90
7.4.1 硬件设计 .....	90
7.4.2 软件设计 .....	90
7.4.3 下载验证 .....	97
<b>第八章 原子云平台连接.....</b>	<b>98</b>
8.1 原子云工作流程 .....	98
8.2 原子云连接实验 .....	99
8.2.1 硬件设计 .....	99
8.2.2 软件设计 .....	99
8.2.3 下载验证 .....	103

## 第一章 全硬件的 TCP/IP 协议栈简介

全硬件 TCP/IP 协议栈芯片采用了“TCP/IP Offload Platform”技术，囊括了 TCP/IP 协议栈全部的四层结构，独立于 MCU 运作，信息的进栈/出栈，封包/解包等网络数据处理全部在全硬件 TCP/IP 网络芯片中进行，高速硬件化 TCP/IP 协议处理卸载掉了 MCU 对于 Ethernet 庞大数据处理的负载，从而，使 MCU 保持高效运转且实现高速实际网络传输。同时，这也避免了 MCU 受到网络攻击的危险，网络攻击不会对 MCU 中的主程序产生影响，增加了 MCU 工作的安全性。大大优化了 MCU 的网络功能，尤其对于不能支持 OS 的 8 bit & 16 bit MCU 的优化提升无疑是革命性的。工程师不需深入了解 TCP/IP 协议，而且程序的烧写和移植比较方便，可以大大的缩短产品开发时间。本教程是针对正点原子战舰开发板板载的以太网芯片配套例程，这个以太网芯片采用了“TCP/IP Offload Platform”技术，实现了以太网通讯。

本章我们分为几个部分讲解：

- 1.1 以太网接入单片机方案
- 1.2 以太网接口芯片 CH395Q 简介
- 1.3 以太网接口芯片 CH395Q 命令简介
- 1.4 以太网接口芯片 CH395Q 寄存器配置与使用
- 1.5 移植 CH395Q 源码

### 1.1 以太网接入单片机方案

在网络项目中，不同的 MCU 实现网络接口通信的方式是有所不同的。根据网络接口通信方式的不同归结为两类方案：第一类是传统的软件 TCP/IP 协议栈方案；第二类为硬件 TCP/IP 协议栈方案。这两种接入方案的介绍如下所示：

#### 1. 传统的软件 TCP/IP 协议栈以太网接入方案

这种方案由 MCU (MAC)+PHY (芯片) 实现以太网物理连接，例如：正点原子的探索者、阿波罗、北极星以及电机开发板都是采用这类型的以太网接入方案，该方案的连接示意图如下图所示：



图 1.1.1 MAC+PHY 以太网方案示意图

可以看到，这种方案就是让 MCU 内嵌了一个 MAC 内核，该内核相当于 TCP/IP 协议栈的数据链路层；板载的 PHY 芯片相当于 TCP/IP 协议栈的物理层；而 lwIP 协议栈实现了应用层、传输层和网络层功能。至于 lwIP 相关的知识，请读者观看正点原子的《lwIP 开发指南》文档。

接下来笔者带大家来了解一下传统软件 TCP/IP 协议栈方案的优缺点，如下所示：

**优点：**

- ① 移植性：可在不同平台、不同编译环境的程序代码经过修改转移到自己的系统中运行。
- ② 可造性：可在 TCP/IP 协议栈的基础上添加和删除相关功能。
- ③ 可扩展性：可扩展到其他领域的应用及开发。

**缺点：**

- ① 内存方面分析：传统的 TCP/IP 方案是移植一个 lwIP 的 TCP/IP 协议 (RAM 50K+, ROM



80K+)，造成主控可用内存减小。

- ② 从代码量分析：移植 lwIP 可能需要的代码量超过 40KB，对于有些主控芯片内存匮乏来说无疑是一个严重的问题。
- ③ 从运行性能方面分析：由于软件 TCP/IP 协议栈方案在通信时候是不断地访问中断机制，造成线程无法运行，如果多线程运行，会使 MCU 的工作效率大大降低。
- ④ 从安全性方面分析：软件协议栈会很容易遭受网络攻击，造成单片机瘫痪。

## 2. 硬件 TCP/IP 协议栈以太网接入方案

所谓全硬件 TCP/IP 协议栈是将传统的软件协议 TCP/IP 协议栈用硬件化的逻辑门电路来实现。芯片内部完成 TCP、UDP、ICMP 等多种应用层协议，并且实现了物理层以太网控制（MAC+PHY）、内存管理等功能，完成了一整套硬件化的以太网解决方案。该方案的连接示意图如下图所示：



图 1.1.2 硬件协议栈连接示意图

可以看到，MCU 可以不具备内嵌的 MAC 控制器而实现以太网连接，这种方式可减少程序员对 TCP/IP 协议的了解，甚至弥补了网络协议安全性不足的短板。全硬件 TCP/IP 协议栈方案的优缺点，如下所示：

### 优点：

- ① 从代码量方面来看：相比于传统的接入已经大大减少了代码量。
- ② 从运行方面来看：极大的减少了中断次数，让单片机更好的完成其他线程的工作。
- ③ 从安全性方面来看：硬件化的逻辑门电路来处理 TCP/IP 协议是不可被攻击的，也就是说网络攻击和病毒对它无效，这也充分弥补了网络协议安全性不足的短板。

### 缺点：

- ① 从可扩展性来看：虽然该芯片内部使用逻辑门电路来实现应用层和物理层协议，但是它具有功能局限性，例如给 TCP/IP 协议栈添加一个协议，这样它无法快速添加了。
- ② 从收发速率来看：全硬件 TCP/IP 协议栈芯片都是采用并口、SPI 以及 IIC 等通讯接口来收发数据，这些数据会受通信接口的速率而影响。

总的来说：全硬件 TCP / IP 协议栈简化传统的软件 TCP / IP 协议栈，卸载了 MCU 用于处理 TCP / IP 这部分的线程，节约 MCU 内部 ROM 等硬件资源，工程师只需进行简单的套接字编程和少量的寄存器操作即可方便地进行嵌入式以太网上层应用开发，减少产品开发周期，降低开发成本。

## 1.2 以太网接口芯片 CH395Q 简介

CH395Q 是南京沁恒微电子推出的一款高性能以太网芯片，正点原子战舰板载的以太网芯片是以南京沁恒微电子股份有限公司的 CH395Q 以太网协议栈管理芯片为核心，该芯片自带了 10/100M 以太网介质传输层（MAC）和物理层（PHY），并完全兼容 IEEE802.3 10/100M 协议，内置了 UDP、TCP 等以太网协议栈的固件，用于单片机系统进行以太网通讯。它支持间接并行总线和高速 SPI 接口 2 种方式与主机进行通信。其内部还集成了以太网数据链路层（MAC）和 10Base-T/100Base-T 以太网物理层（PHY），支持自动协商（10/100-基于全双工/半双工）。与传统软件协议栈不同，CH395Q 内嵌的 8 个独立硬件套接字可以进行 8 路独立通信，该 8 路 socket 的通信效率互不影响，使用起来十分方便。

CH395Q 以太网协议栈管理芯片的整体框图如下图所示：

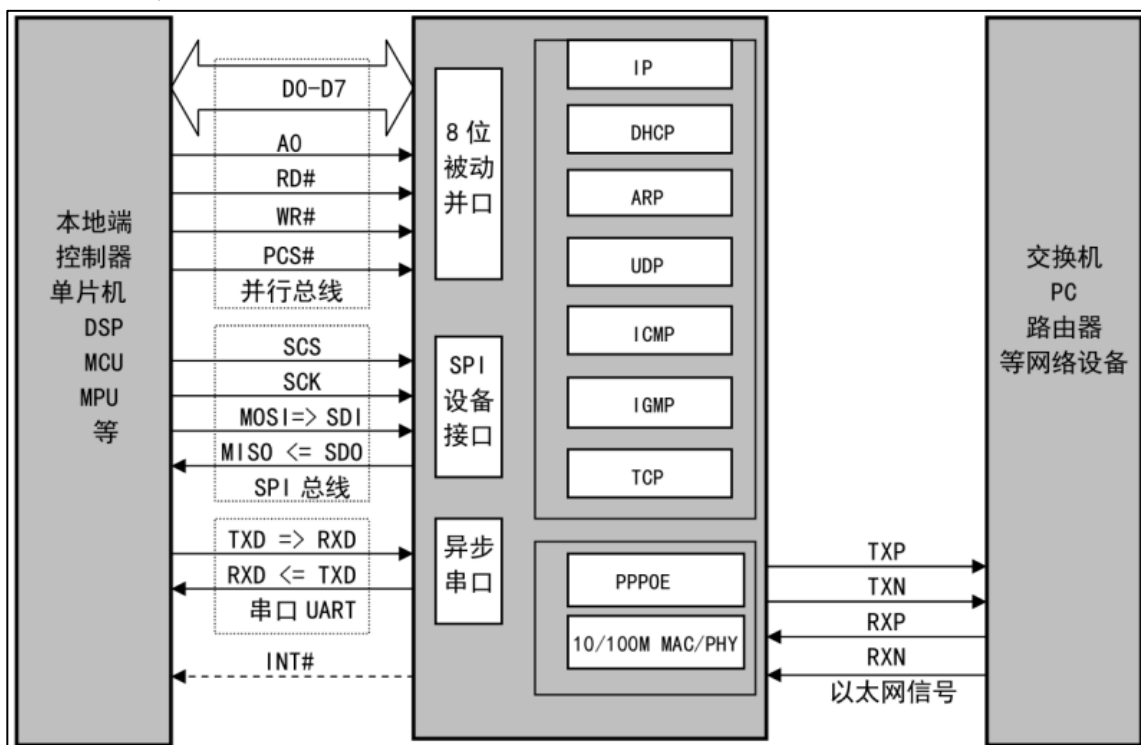


图 1.2.1 CH395Q 应用框图

可以看出，CH395Q 芯片具备三种通信接口，它内嵌 TCP/IP 协议栈并且在其基础上实现了各层间的子协议。本教程的配套例程是使用 SPI 通信接口，所以 MCU 和 CH395Q 芯片交互的引脚数量只有 SCS、SCK、MOSI、MISO、RTS 以及 INT#。下面笔者使用一个示意图来总结本节的内容如下图所示：

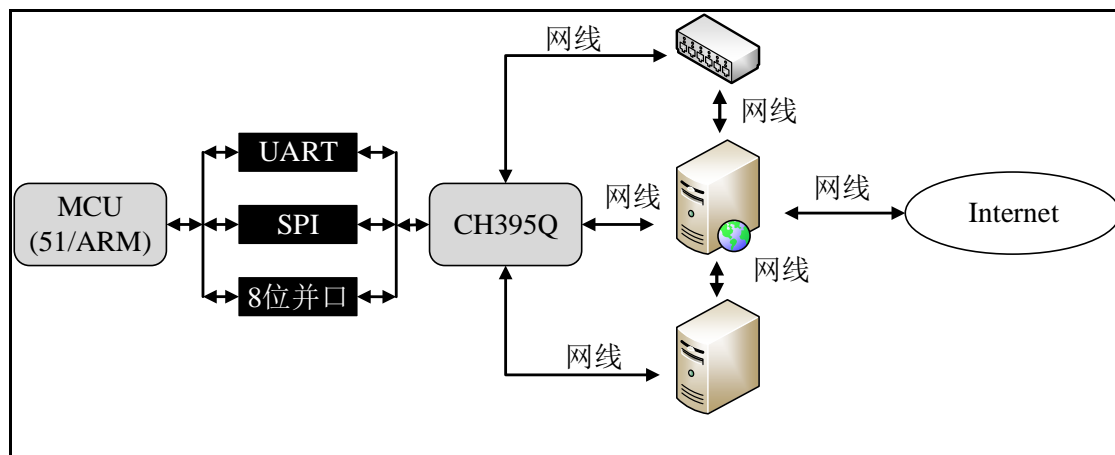


图 1.2.2 CH395Q 连接图

从上图可以看出，CH395Q 以太网芯片类似于网卡，我们可通过普通的接口发送数据至 CH395Q 以太网芯片当中，这些数据经过以太网芯片处理发送至网络当中。

### CH395Q 特性参数

CH395Q 的各项基本参数，如下表所示：



项目	说明
通讯接口	SPI/UART/8 位并口
通讯接口速率	30MHz（最高）
配置方式	命令
工作模式	UDP、TCP 客户端、TCP 服务器
物理层支持	支持 10/100M，全双工/半双工自适应，兼容 IEEE802.3 协议
TCP/IP 协议簇支持	支持 IPv4、DHCP、ARP、ICMP、IGMP、UDP、TCP 协议
PPPOE 协议支持	支持 PAP、CHAP 认证
Socket 支持	提供 8 个独立的 Socket 接口，可同时进行数据收发
收发缓冲区	内置 24KB 用于以太网数据收发的 RAM，每个 Socket 收发缓冲区可自由配置
EEPROM	内置 4KB EEPROM

表 1.2.1 CH 395Q 模块基本参数

CH 395Q 模块的各项电气参数，如下表所示：

项目	说明
电源电压	3.3V
IO 口电平	3.3V

表 1.2.2 CH395Q 模块电气参数

### 1.3 以太网接口芯片 CH395Q 命令简介

CH395Q 芯片通过 SPI 接口与外接控制器进行数据通讯，外接控制器需要通过命令控制 CH395Q 芯片和与 CH395Q 芯片进行数据交互，CH395Q 芯片支持的命令有很多，具体的命令和命令详述请见 CH395Q 芯片的手册《CH395DS1.PDF》，本文仅介绍几个常用的命令，如下表所示：

命令码	命令名称	命令用途
0x01	CMD_GET_IC_VER	获取芯片及固件版本
0x05	CMD_RESET_ALL	执行硬件复位
0x06	CMD_CHECK_EXIST	测试通讯接口和工作方式
0x19	CMD_GET_GLOB_INT_STATUS_ALL	获取全局中断状态
0x21	CMD_SET_MAC_ADDR	设置 MAC 地址
0x22	CMD_SET_IP_ADDR	设置 IP 地址
0x23	CMD_SET_GWIP_ADDR	设置网关 IP 地址
0x24	CMD_SET_MASK_ADDR	设置子网掩码
0x26	CMD_GET_PHY_STATUS	获取 PHY 的状态
0x27	CMD_INIT_CH395	初始化 CH395 芯片
0x2C	CMD_GET_CMD_STATUS	获取命令执行状态
0x30	CMD_GET_INT_STATUS_SN	获取 Socket 的中断状态
0x31	CMD_SET_IP_ADDR_SN	设置 Socket 的目的 IP 地址
0x32	CMD_SET_DES_PORT_SN	设置 Socket 的目的端口
0x33	CMD_SET_SOUR_PORT_SN	设置 Socket 的源端口
0x34	CMD_SET_PROTO_TYPE_SN	设置 Socket 的工作模式

0x35	CMD_OPEN_SOCKET_SN	打开 Socket
0x36	CMD_TCP_LISTEN_SN	启动 Socket 监听
0x37	CMD_TCP_CONNECT_SN	启动 Socket 连接
0x38	CMD_TCP_DISCONNECT_SN	断开 Socket 的 TCP 连接
0x39	CMD_WRITE_SEND_BUF_SN	向 Socket 发送缓冲区写数据
0x3B	CMD_GET_RECV_LEN_SN	获取 Socket 接收数据长度
0x3C	CMD_READ_RECV_BUF_SN	从 Socket 接收缓冲区接收数据
0x3D	CMD_CLOSE_SOCKET_SN	关闭 Socket
0x41	CMD_DHCP_ENABLE	启动（停止）DHCP
0x42	CMD_GET_DHCP_STATUS	获取 DHCP 状态
0x43	CMD_GET_IP_INF	获取 IP、MASK、DNS 等信息

表 1.3.1 CH395Q 芯片常用命令

#### 1、CMD\_GET\_IC\_VER——获取芯片及固件版本

该命令用于获取 CH395 的芯片及固件版本。该命令会返回 1 个字节的版本号数据，其中位 7 为 0、位 6 为 1、位 5~0 为版本号。

#### 2、CMD\_RESET\_ALL——执行硬件复位

该命令使 CH395 芯片执行硬件复位。通常情况下，硬件复位在 50ms 时间之内完成。

#### 3、CMD\_CHECK\_EXIST——测试通讯接口和工作方式

该命令用于测试通讯接口和工作状态。该命令需要 1 个字节的任意数据，如果 CH395 正常工作且通讯接口无误，那么将会返回 1 个字节的的数据，返回的值为输入数据按位取反的结果。

#### 4、CMD\_GET\_GLOB\_INT\_STATUS\_ALL——获取全局中断状态

该命令用于获取全局中断状态。该命令发送完成时会返回 2 个字节的全球中断状态，其定义如下表所示：

数据位	名称	描述
[18:12]	-	保留
11	GINT_STAT SOCK7	Socket7 中断
10	GINT_STAT SOCK6	Socket6 中断
9	GINT_STAT SOCK5	Socket5 中断
8	GINT_STAT SOCK4	Socket4 中断
7	GINT_STAT SOCK3	Socket3 中断
6	GINT_STAT SOCK2	Socket2 中断
5	GINT_STAT SOCK1	Socket1 中断
4	GINT_STAT SOCK0	Socket0 中断
3	GINT_STAT_DHCP 和 GINT_STAT_PPPOE	DHCP 和 PPPOE 中断
2	GINT_STAT_PHY_CHANGE	PHY 状态改变
1	GINT_STAT_IP_CONFLI	IP 冲突
0	GINT_STAT_UNREACH	不可达中断

表 1.3.2 全局中断状态定义

**GINT\_STAT\_UNREACH:** 不可达中断。当 CH395 芯片收到 ICMP 不可达中断报文后，将不可达 IP 数据包的 IP 地址、端口、协议类型保存到不可达信息表中，然后产生此中断。

**GINT\_STAT\_IP\_CONFLI:** IP 冲突中断。当 CH395 芯片检测到自身 IP 地址和同一网段内

的其他网络设备 IP 地址相同时，会产生此中断。

**GINT\_STAT\_PHY\_CHANGE:** PHY 变化中断。当 CH395 芯片的 PHY 连接变化时产生此中断。

**GINT\_STAT\_DHCP 和 GINT\_STAT\_PPPOE:** DHCP 中断和 PPPOE 中断共用此中断源。如果外部主控使能了 CH395 的 DHCP 功能或 PPPOE 功能，CH395 将会产生此中断。

**GINT\_STAT\_SOCKET0~GINT\_STAT\_SOCKET7:** Socket 中断。当 Socket 有中断事件时，CH395 芯片会产生此中断。

此命令执行完毕后，CH395 芯片会自动将 INT 引脚置为高电平并清除全局中断。

#### 5、CMD\_SET\_MAC\_ADDR——设置 MAC 地址

该命令用于设置 ATK-MO395Q 模块的 MAC 地址。该命令需要输入 6 个字节的 MAC 地址数据（MAC 地址低字节在前），CH395 芯片会将该 MAC 地址保存到内部的 EEPROM 中，该命令需要 100ms 的执行时间。

注意：CH395 芯片出厂时已经烧录了由 IEEE 分配的 MAC 地址，如非必要请勿设置 MAC 地址。

#### 6、CMD\_SET\_IP\_ADDR——设置 IP 地址

该命令用于设置 ATK-MO395Q 模块的 IP 地址。该命令需要输入 4 个字节的 IP 地址数据（IP 地址低字节在前）。

#### 7、CMD\_SET\_GWIP\_ADDR——设置网关 IP 地址

该命令用于设置 ATK-MO395Q 模块的网关 IP 地址。该命令需要输入 4 个字节的网关 IP 地址（网关 IP 地址低字节在前）。

#### 8、CMD\_SET\_MASK\_ADDR——设置子网掩码

该命令用于设置 ATK-MO395Q 模块的子网掩码。该命令需要输入 4 字节的子网掩码数据（子网掩码低字节在前）。

#### 9、CMD\_GET\_PHY\_STATUS——获取 PHY 的状态

该命令用于获取 ATK-MO395Q 模块的 PHY 状态。该命令会返回 1 个字节的 PHY 状态码数据，其定义如下表所示：

PHY 状态码	描述
0x01	PHY 连接断开
0x02	PHY 连接为 10M 全双工
0x04	PHY 连接为 10M 半双工
0x08	PHY 连接为 100M 全双工
0x10	PHY 连接为 100M 半双工

表 1.3.3 PHY 状态码定义

#### 10、CMD\_INIT\_CH395——初始化 CH395 芯片

该命令用于初始化 CH395 芯片，初始化的内容包括 MAC、PHY 和 TCP/IP 协议栈，该命令约需要 350ms 的执行时间。

#### 11、CMD\_GET\_CMD\_STATUS——获取命令执行状态

该命令用于获取命令的执行状态。该命令发送完成时会返回 1 个字节的状态码数据，其定义如下表所示：

状态码	名称	描述
0x00	CH395_ERR_SUCCESS	成功
0x10	CH395_ERR_BUSY	忙，表示命令正在执行
0x11	CH395_ERR_MEM	内存管理错误

0x12	CH395_ERR_BUF	缓冲区错误
0x13	CH395_ERR_TIMEOUT	超时
0x14	CH395_ERR_RET	路由错误
0x15	CH395_ERR_ABRT	连接中止
0x16	CH395_ERR_RST	连接复位
0x17	CH395_ERR_CLSD	连接关闭
0x18	CH395_ERR_CONN	无连接
0x19	CH395_ERR_VAL	值错误
0x1A	CH395_ERR_ARG	参数错误
0x1B	CH395_ERR_USE	已被使用
0x1C	CH395_ERR_IF	MAC 错误
0x1D	CH395_ERR_ISCONN	已连接
0x20	CH395_ERR_OPEN	已打开

表 1.3.4 命令执行状态码定义

若外部控制器收到 CH395\_ERR\_BUSY 的返回值，表示 CH395 正在执行命令，外部主控应延时 2ms 以上再获取命令的执行状态。

## 12、CMD\_GET\_INT\_STATUS\_SN——获取 Socket 的中断状态

该命令用于获取 Socket 的中断状态。该命令需要输入 1 个字节的 Socket 标号，该命令会返回 1 个字节的 Socket 中断状态码，其定义如下所示：

Socket 中断状态码位	名称	描述
7	-	保留
6	SINT_STAT_TIM_OUT	超时
5	-	保留
4	SINT_STAT_DISCONNECT	TCP 断开
3	SINT_STAT_CONNECT	TCP 连接
2	SINT_STAT_RECV	接收缓冲区非空
1	SINT_STAT_SEND_OK	发送成功
0	SINT_STAT_SENBUF_FREE	发送缓冲区空闲

表 1.3.5 Socket 中断状态码定义

**SINT\_STAT\_SENBUF\_FREE:** 发送缓冲区非空中断。外部控制器向 Socket 发送换从去写入数据后，需等待该中断产生，才能再次向 Socket 发送缓冲区写入数据。

**SINT\_STAT\_SEND\_OK:** 发送成功中断。当数据包被成功发送后，会产生此中断。外部控制器向 Socket 的发送缓冲区写入数据后，CH395 可能会将数据封装成若干个数据包，每成功发送一个数据包，都会产生一次该中断。

**SINT\_STAT\_RECV:** 接收缓冲区非空中断。当 Socket 接收到数据时，会产生该中断。

**SINT\_STAT\_CONNECT:** TCP 连接中断。该中断仅在 TCP 模式下有效，表明 TCP 连接成功，外部控制器必须等待该中断产生才能进行 TCP 数据传输。

**SINT\_STAT\_DISCONNECT:** TCP 连接断开中断。该中断仅在 TCP 模式下有效，表明 TCP 连接断开。

**SINT\_STAT\_TIM\_OUT:** 超时中断。在 TCP 模式下，TCP 连接、断开、发送数据等过程中出现超时，则会产生此中断；IPRAW 和 UDP 模式下，发送数据失败也会产生此中断。

## 13、CMD\_SET\_IP\_ADDR\_SN——设置 Socket 的目的 IP 地址

该命令用于设置 Socket 的目的 IP 地址。该命令需要输入 1 个字节的 Socket 标号和 4 个字

节的目的 IP 地址数据（目的 IP 地址低字节在前）。

#### 14、CMD\_SET\_DEST\_PORT\_SN——设置 Socket 的目的端口

该命令用于设置 Socket 的目的端口。该命令需要输入 1 个字节的 Socket 标号和 2 个字节的端口号数据（端口号低字节在前）。

#### 15、CMD\_SET\_SOUR\_PORT\_SN——设置 Socket 的源端口

该命令用于设置 Socket 的原端口。该命令需要输入 1 个字节的 Socket 标号和 2 个字节的源端口号数据（源端口号低字节在前）。

#### 16、CMD\_SET\_PROTO\_TYPE\_SN——设置 Socket 的工作模式

该命令用于设置 Socket 的工作模式。该命令需要输入 1 个字节的 Socket 标号和 1 字节的工作模式码，其定义如下所示：

工作模式码	名称	描述
0x03	PROTO_TYPE_TCP	TCP 模式
0x02	PROTO_TYPE_UDP	UDP 模式
0x01	PROTO_TYPE_MAC_RAW	MAC 原始报文模式
0x00	PROTO_TYPE_IP_RAW	IP 原始报文模式

表 1.3.6 Socket 工作模式码定义

#### 17、CMD\_OPEN\_SOCKET\_SN——打开 Socket

该命令用于打开 Socket。该命令需要输入 1 个字节的 Socket 标号。

#### 18、CMD\_TCP\_LISTEN\_SN——启动 Socket 监听

该命令用于使能 Socket 进入监听模式（即 TCP Server 模式），该命令仅在 TCP 模式下有效。该命令需要输入 1 个字节的 Socket 标号。

#### 19、CMD\_TCP\_CONNECT\_SN——启动 Socket 连接

该命令用于使能 Socket 进入连接模式（即 TCP Client 模式），该命令仅在 TCP 模式下有效。该命令需要输入 1 个字节的 Socket 标号。

#### 20、CMD\_TCP\_DISCONNECT\_SN——断开 Socket 的 TCP 连接

该命令用于断开 Socket 当前的 TCP 连接，该命令仅在 TCP 模式下有效。该命令需要输入 1 个字节的 Socket 标号。

#### 21、CMD\_WRITE\_SEND\_BUF\_SN——向 Socket 发送缓冲区写数据

该命令用于向 Socket 的发送缓冲区写入数据。该命令需要输入 1 个字节的 Socket 标号、2 个字节的待写入数据长度（低字节在前）和若干字节的数据流。

#### 22、CMD\_GET\_RECV\_LEN\_SN——获取 Socket 接收数据长度

该命令用于获取当前接收缓冲区的有效数据长度。该命令需要输入 1 个字节的 Socket 标号，该命令会返回 2 字节的有效数据长度数据（低字节在前）。

#### 23、CMD\_READ\_RECV\_BUF\_SN——从 Socket 接收缓冲区接收数据

该命令用于从 Socket 接收缓冲区读取数据。该命令需要输入 1 个字节的 Socket 标号和 2 个字节的读取长度（低字节在前），该命令会返回若干字节的数据流。

#### 24、CMD\_CLOSE\_SOCKET\_SN——关闭 Socket

该命令用于关闭 Socket。该命令需要输入 1 个字节的 Socket 标号。

#### 25、CMD\_DHCP\_ENABLE——启动（停止）DHCP

该命令用于启动或停止 DHCP。该命令需要输入 1 个字节的 DHCP 使能码，其定义如下表所示：

DHCP 使能码	描述
0	关闭 DHCP
1	启动 DHCP

表 1.3.7 DHCP 使能码

## 26、CMD\_GET\_DHCP\_STATUS——获取 DHCP 状态

该命令用于获取 DHCP 的状态。该命令发送完成时会返回 1 个字节的 DHCP 状态码，其定义如下表所示：

DHCP 状态码	描述
0	成功
1	错误

表 1.3.8 DHCP 状态码

## 27、CMD\_GET\_IP\_INF——获取 IP、MASK、DNS 等信息

该命令用于获取 IP 地址、网关 IP 地址、子网掩码、DNS 等信息。该命令会一次返回 20 个字节数据，分别为 4 个字节 IP 地址、4 个字节网关 IP 地址、4 个字节子网掩码、4 个字节 DNS 服务器 1 地址、4 个字节 DNS 服务器 2 地址。

## 1.4 以太网接口芯片 CH395Q 寄存器配置与使用

CH395Q 以太网芯片是使用命令来配置以太网环境，通过通讯接口把配置命令发送至 CH395Q 芯片当中，CH395Q 以太网芯片会根据接收的配置命令使能相应的功能。沁恒微电子为 CH395Q 以太网芯片配置了一套驱动文件，这些文件定义了配置命令和配置函数，同时，它还提供了例程源码方便用户学习。该驱动文件下载流程如下所示：

- ① 打开沁恒微电子官网网址。
  - ② 在官方网站搜索 CH395Q，查找完成之后下载 CH395Q 使用文档和例程源码。
- 下载完成之后我们可以得到以下压缩包。

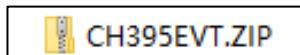


图 1.4.1 CH395Q 官方例程

通过对 CH395EVT.ZIP 进行解压可得到以下文件和文件夹，如下图所示：

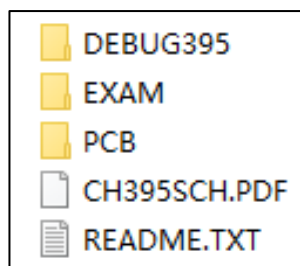


图 1.4.2 解压后的例程文件

从上图中，笔者重点讲解 EXAM\PCB 路径下的 CH395CMD.C/H 和 CH395INC.H 文件，这些文件就是定义了 CH395Q 支持的配置命令和配置函数。配置命令是在 CH395INC.H 文件下定义的，而配置函数（CH395CMD.C 文件下定义）就是对配置命令进行封装的，然后通过通讯接口函数发送至 CH395Q 以太网芯片当中。CH395CMD.H 文件主要声明 CH395CMD.C 文件下的配置函数，提供外部文件使用。

笔者为了兼容正点原子最新的代码格式规范，我们把 CH395CMD.C/H 和 CH395INC.H 文件进行了修改，主要对这些文件内的变量、函数及注释格式做了全新的改编。



根据上述内容的要求,笔者把 CH395CMD.C/H 文件名修改为 ch395cmd.c/h,而 CH395INC.H 文件名修改为 ch395inc.h 文件(寄存器命令定义文件),下面我们分别地讲解这些文件定义的内容。

### 1. 文件 ch395inc.h

该文件定义了 CH395Q 以太网芯片支持的配置命令,我们可以根据它提供的命令来配置以太网芯片的功能,这些配置命令如下源码所示:

```
#define CMD01_GET_IC_VER          0x01    /* 获取芯片以及固件版本号 */
#define CMD31_SET_BAUDRATE        0x02    /* 串口方式 */
#define CMD00_ENTER_SLEEP         0x03    /* 进入睡眠状态 */
#define CMD00_RESET_ALL           0x05    /* 执行硬件复位 */
#define CMD11_CHECK_EXIST         0x06    /* 测试通讯接口以及工作状态 */
#define CMD02_GET_GLOB_INT_STATUS_ALL 0x19  /* 获取全局中断状态 */
#define CMD10_SET_PHY             0x20    /* 设置 PHY, 自动协商 */
#define CMD60_SET_MAC_ADDR        0x21    /* 设置 MAC 地址 */
#define CMD40_SET_IP_ADDR         0x22    /* 设置 IP 地址 */
#define CMD40_SET_GWIP_ADDR       0x23    /* 设置网关 IP 地址 */
#define CMD40_SET_MASK_ADDR       0x24    /* 设置子网掩码 */
#define CMD90_SET_MAC_FILT        0x25    /* 设置 MAC 过滤 */
#define CMD01_GET_PHY_STATUS      0x26    /* 获取 PHY 当前状态 */
#define CMD0W_INIT_CH395          0x27    /* 初始化 CH395 */
#define CMD08_GET_UNREACH_IPPORT  0x28    /* 获取不可达信息 */
#define CMD01_GET_GLOB_INT_STATUS 0x29    /* 获取全局中断状态 */
#define CMD10_SET_RETRAN_COUNT    0x2A    /* 重试次数 */
#define CMD20_SET_RETRAN_PERIOD   0x2B    /* 重试周期 */
#define CMD01_GET_CMD_STATUS      0x2C    /* 获取命令执行状态 */
#define CMD06_GET_REMOT_IPP_SN    0x2D    /* 获取远端的端口以及 IP 地址 */
#define CMD10_CLEAR_RECV_BUF_SN   0x2E    /* 清除接收缓冲区 */
#define CMD12_GET_SOCKET_STATUS_SN 0x2F    /* 获取 socket n 状态 */
#define CMD11_GET_INT_STATUS_SN   0x30    /* 获取 socket n 的中断状态 */
#define CMD50_SET_IP_ADDR_SN      0x31    /* 设置 socket n 的目的 IP 地址 */
#define CMD30_SET_DEST_PORT_SN    0x32    /* 设置 socket n 的目的端口 */
#define CMD30_SET_SOUR_PORT_SN    0x33    /* 设置 socket n 的源端口 */
#define CMD20_SET_PROTO_TYPE_SN   0x34    /* 设置 socket n 的协议类型 */
#define CMD1W_OPEN_SOCKET_SN      0x35    /* 打开 socket n */
#define CMD1W_TCP_LISTEN_SN       0x36    /* socket n 监听 */
#define CMD1W_TCP_CONNECT_SN      0x37    /* socket n 连接 */
#define CMD1W_TCP_DISCONNECT_SN   0x38    /* socket n 断开连接, */
#define CMD30_WRITE_SEND_BUF_SN   0x39    /* 向 socket n 缓冲区写入数据 */
#define CMD12_GET_RECV_LEN_SN     0x3B    /* 获取 socket n 接收数据的长度 */
#define CMD30_READ_RECV_BUF_SN    0x3C    /* 读取 socket n 接收缓冲区数据 */
#define CMD1W_CLOSE_SOCKET_SN     0x3D    /* 关闭 socket n */
#define CMD20_SET_IPRAW_PRO_SN    0x3E    /* 在 IP RAW 下 */
#define CMD01_PING_ENABLE         0x3F    /* 开启/关闭 PING */
```

```

#define CMD06_GET_MAC_ADDR          0x40    /* 获取 MAC 地址 */
#define CMD10_DHCP_ENABLE            0x41    /* DHCP 使能 */
#define CMD01_GET_DHCP_STATUS        0x42    /* 获取 DHCP 状态 */
#define CMD014_GET_IP_INF             0x43    /* IP, 子网掩码, 网关 */
#define CMD00_PPPOE_SET_USER_NAME     0x44    /* 设置 PPPOE 用户名 */
#define CMD00_PPPOE_SET_PASSWORD     0x45    /* 设置密码 */
#define CMD10_PPPOE_ENABLE            0x46    /* PPPOE 使能 */
#define CMD01_GET_PPPOE_STATUS        0x47    /* 获取 pppoe 状态 */
#define CMD20_SET_TCP_MSS             0x50    /* 设置 TCP MSS */
#define CMD20_SET_TTL                 0x51    /* 设置 TTL, TTL 最大值为 128 */
#define CMD30_SET_RECV_BUF            0x52    /* 设置 SOCKET 接收缓冲区 */
#define CMD30_SET_SEND_BUF            0x53    /* 设置 SOCKET 发送缓冲区 */
#define CMD10_SET_MAC_RECV_BUF        0x54    /* 设置 MAC 接收缓冲区 */
#define CMD40_SET_FUN_PARA            0x55    /* 设置功能参数 */
#define CMD40_SET_KEEP_LIVE_IDLE      0x56    /* 设置 KEEPLIVE 空闲 */
#define CMD40_SET_KEEP_LIVE_INTVL     0x57    /* 设置间隔时间 */
#define CMD10_SET_KEEP_LIVE_CNT       0x58    /* 重试次数 */
#define CMD20_SET_KEEP_LIVE_SN        0x59    /* 设置 socket nkeepalive 功能 */
#define CMD00_EEPROM_ERASE            0xE9    /* 擦除 EEPROM */
#define CMD30_EEPROM_WRITE            0xEA    /* 写 EEPROM */
#define CMD30_EEPROM_READ             0xEB    /* 读 EEPROM */
#define CMD10_READ_GPIO_REG           0xEC    /* 读 GPIO 寄存器 */
#define CMD20_WRITE_GPIO_REG          0xED    /* 写 GPIO 寄存器 */

```

相关命令描述请看本章节的“以太网接口芯片 CH395Q 命令简介”小节。

## 2. 文件 ch395cmd.c

该文件的函数是根据 CH395Q 以太网配置命令编写的,下面笔者挑几个重要的函数来讲解,如下源码所示:

### (1) 函数 ch395\_cmd\_reset

该函数的作用是硬件复位 CH395Q 以太网芯片, 如下源码所示:

```

/**
 * @brief      复位 ch395 芯片
 * @param      无
 * @retval     无
 */
void ch395_cmd_reset(void)
{
    ch395_write_cmd(CMD00_RESET_ALL);
    ch395_scs_hign;
}

```

此函数通过发送 CMD00\_RESET\_ALL 命令硬件复位 CH395Q 以太网芯片。

### (2) 函数 ch395\_cmd\_check\_exist

该函数的作用是测试 MUC 与 CH395Q 以太网芯片是否正常通讯, 如下源码所示:

```

/**

```

```

* @brief      测试命令，用于测试硬件以及接口通讯，
* @param      1 字节测试数据
* @retval     硬件 ok，返回 testdata 按位取反
*/
uint8_t ch395_cmd_check_exist(uint8_t testdata)
{
    uint8_t i;

    ch395_write_cmd(CMD11_CHECK_EXIST);
    ch395_write_data(testdata);
    i = ch395_read_data();
    ch395_scs_hign;
    return i;
}

```

此函数把 CMD11\_CHECK\_EXIST 命令和测试数据发送至 CH395Q 以太网芯片中，发送完成之后接收 CH395Q 以太网芯片返回的数据，如果 MCU 接收的数据是发送数据的反码，则表示通信正常。

### 3. 文件 ch395cmd.h

声明 ch395cmd.c 文件下的配置函数，这些声明函数如下源码所示：

```

#ifndef __CH395CMD_H__
#define __CH395CMD_H__
#include "../BSP/CH395Q/ch395inc.h"
#include "../SYSTEM/sys/sys.h"

/* 复位 */
void ch395_cmd_reset(void);
/* 睡眠 */
void ch395_cmd_sleep(void);
/* 获取芯片及固件版本号 */
uint8_t ch395_cmd_get_ver(void);
/* 测试命令 */
uint8_t ch395_cmd_check_exist(uint8_t testdata);
/* 设置 phy 状态 */
void ch395_cmd_set_phy(uint8_t phystat);
/* 获取 phy 状态 */
uint8_t ch395_cmd_get_phy_status(void);

/*****省略部分函数*****/

#endif

```

该文件非常简单，它主要声明 ch395cmd.c 文件下的配置函数，提供给其他文件调用。

至此，笔者已经讲解完了官方提供的 ch395cmd.c/h 文件和 ch395inc.h 文件。接下来笔者将讲解官方提供的另外一个重要文件，它们的文件名为 CH395.C/H 文件，这些文件在 EXAM\

EXAMx (x: 0~15) 文件夹找到, 是官方提供给用户的例程源码, 主要作用是调用 ch395cmd.c 文件下的配置函数初始化 CH395Q 以太网芯片和配置以太网环境, 例如: UDP、TCP、ICMP 等协议。这里笔者会和前面一样对它们进行代码格式修改, 首先我们把 CH395.C/H 文件名修改为 ch395.c/h 文件, 接着在 ch395.c 文件下封装了 ch395\_write\_cmd、ch395\_read\_data 以及 ch395\_write\_data 函数, 这些函数都是调用了 SPI1 接口函数发送数据和命令。ch395.c/h 的文件结构, 如下所示:

#### 4. 文件 ch395.c

##### (1) 声明头文件

```
#include "../BSP/CH395Q/ch395.h"
#include "../SYSTEM/delay/delay.h"
#include "../BSP/LCD/lcd.h"
#include "../BSP/SPI/spi.h"
```

##### (2) 定义网络配置结构体

```
struct ch395q_t g_ch395q_sta;
```

该结构体主要保存网络相关的参数及回调函数。

##### (3) CH395Q 引脚初始化

```
/**
 * @brief      ch395_gpio 初始化
 * @param      无
 * @retval     无
 */
void ch395_gpio_init( void )
{
    GPIO_InitTypeDef gpio_init_struct;

    CH395_SCS_GPIO_CLK_ENABLE();    /* 使能 SCS 时钟 */
    CH395_INT_GPIO_CLK_ENABLE();    /* 使能 INT 时钟 */
    CH395_RST_GPIO_CLK_ENABLE();    /* 使能 RST 时钟 */

    /* SCS */
    gpio_init_struct.Pin = CH395_SCS_GPIO_PIN;
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_MEDIUM;
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;    /* 推拉输出 */
    HAL_GPIO_Init( CH395_SCS_GPIO_PORT, &gpio_init_struct );

    /* 初始化中断引脚 */
    gpio_init_struct.Pin = CH395_INT_GPIO_PIN;
    gpio_init_struct.Mode = GPIO_MODE_INPUT;    /* 输入 */
    gpio_init_struct.Pull = GPIO_PULLUP;    /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;    /* 高速 */
    HAL_GPIO_Init( CH395_INT_GPIO_PORT, &gpio_init_struct );

    gpio_init_struct.Pin = CH395_RST_GPIO_PIN;
```

```

gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;          /* 输出 */
gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;        /* 高速 */
gpio_init_struct.Pull = GPIO_PULLUP;                  /* 上拉 */
HAL_GPIO_Init( CH395_RST_GPIO_PORT, &gpio_init_struct );
HAL_GPIO_WritePin(CH395_RST_GPIO_PORT, CH395_RST_GPIO_PIN, GPIO_PIN_SET);
delay_ms(20);
}

```

可以看出，上述的源码只初始化了片选、中断和复位引脚，而 SPI 接口我们会在 spi.c 驱动文件中定义。

#### (4) 通信函数

```

/**
 * @brief      硬件 SPI 输出且输入 8 个位数据
 * @param      d:将要送入到 ch395 的数据
 * @retval     无
 */
uint8_t ch395_read_write_byte( uint8_t data )
{
    uint8_t rxdata;
    rxdata = spi1_read_write_byte(data); /* SPI 写入一个 CH395Q 数据并返回一个数据 */
    return rxdata;                       /* 返回收到的数据 */
}

/**
 * @brief      向 ch395 写命令
 * @param      将要写入 ch395 的命令码
 * @retval     无
 */
void ch395_write_cmd( uint8_t mcmd )
{
    ch395_scs_hign;          /* 防止 CS 原来为低，先将 CS 置高 */
    ch395_scs_low;           /* 命令开始，CS 拉低 */
    ch395_read_write_byte(mcmd); /* SPI 发送命令码 */
    delay_ms(2);             /* 必要延时，延时 1.5uS 确保读写周期不小于 1.5uS */
}

/**
 * @brief      向 ch395 写数据
 * @param      将要写入 ch395 的数据
 * @retval     无
 */
void ch395_write_data( uint8_t mdata )
{

```

```

    ch395_read_write_byte(mdata);          /* SPI 发送数据 */
}

/**
 * @brief      从 ch395 读数据
 * @param      无
 * @retval     返回读取的数据
 */
uint8_t ch395_read_data( void )
{
    uint8_t i;
    i = ch395_read_write_byte(0xff);       /* SPI 读数据 */
    return i;
}

```

这些函数最终还是调用了 `spi1_read_write_byte` 函数发送和读取数据。如果读者想了解 SPI 协议，请参考正点原子的《STM32F103 战舰开发指南》第三十六章的内容。

### (5) 保活定时器设置函数

```

/**
 * @brief      ch395_keeplive_set 保活定时器参数设置
 * @param      无
 * @retval     无
 */
void ch395_keeplive_set(void)
{
    ch395_keeplive_cnt(DEF_KEEP_LIVE_CNT);
    ch395_keeplive_idle(DEF_KEEP_LIVE_IDLE);
    ch395_keeplive_intvl(DEF_KEEP_LIVE_PERIOD);
}

```

此函数用于 TCP 协议的客户端，超出规定的时间就会触发中止连接事件。

### (6) 设置 socket 参数

```

/**
 * @brief      ch395 socket 配置
 * @param      ch395_socket: Socket 配置信息
 * @retval     无
 */
uint8_t ch395q_socket_config(ch395_socket * ch395_socket)
{
    if (ch395_socket == NULL)
    {
        return 0;
    }

    if (g_ch395q_sta.dhcp_status == DHCP_UP) /* DHCP 获取成功状态 */

```



```
{
    ch395_socket->net_info.ip[0] = g_ch395q_sta.ipinf_buf[0];
    ch395_socket->net_info.ip[1] = g_ch395q_sta.ipinf_buf[1];
    ch395_socket->net_info.ip[2] = g_ch395q_sta.ipinf_buf[2];
    ch395_socket->net_info.ip[3] = g_ch395q_sta.ipinf_buf[3];

    ch395_socket->net_info.gwip[0] = g_ch395q_sta.ipinf_buf[4];
    ch395_socket->net_info.gwip[1] = g_ch395q_sta.ipinf_buf[5];
    ch395_socket->net_info.gwip[2] = g_ch395q_sta.ipinf_buf[6];
    ch395_socket->net_info.gwip[3] = g_ch395q_sta.ipinf_buf[7];

    ch395_socket->net_info.mask[0] = g_ch395q_sta.ipinf_buf[8];
    ch395_socket->net_info.mask[1] = g_ch395q_sta.ipinf_buf[9];
    ch395_socket->net_info.mask[2] = g_ch395q_sta.ipinf_buf[10];
    ch395_socket->net_info.mask[3] = g_ch395q_sta.ipinf_buf[11];

    ch395_socket->net_info.dns1[0] = g_ch395q_sta.ipinf_buf[12];
    ch395_socket->net_info.dns1[1] = g_ch395q_sta.ipinf_buf[13];
    ch395_socket->net_info.dns1[2] = g_ch395q_sta.ipinf_buf[14];
    ch395_socket->net_info.dns1[3] = g_ch395q_sta.ipinf_buf[15];

    ch395_socket->net_info.dns2[0] = g_ch395q_sta.ipinf_buf[16];
    ch395_socket->net_info.dns2[1] = g_ch395q_sta.ipinf_buf[17];
    ch395_socket->net_info.dns2[2] = g_ch395q_sta.ipinf_buf[18];
    ch395_socket->net_info.dns2[3] = g_ch395q_sta.ipinf_buf[19];
}
else /* DHCP 获取失败状态, 设置静态 IP 地址信息 */
{
    /* 设置 CH395 的 IP 地址 */
    ch395_cmd_set_ipaddr(ch395_socket->net_config.ipaddr);
    /* 设置网关地址 */
    ch395_cmd_set_gw_ipaddr(ch395_socket->net_config.gwipaddr);
    /* 设置子网掩码, 默认为 255.255.255.0 */
    ch395_cmd_set_maskaddr(ch395_socket->net_config.maskaddr);
    ch395_cmd_init();
    delay_ms(100);
}

ch395_cmd_set_macaddr(ch395_socket->net_config.macaddr); /* 设置 MAC 地址 */

memcpy(&g_ch395q_sta.socket[ch395_socket->socket_index].config,
        ch395_socket, sizeof(ch395_socket));
```

```

switch(ch395_socket->proto)
{
    case CH395Q_SOCKET_UDP:
        /* socket 为 UDP 模式 */
        /* 设置 socket 0 目标 IP 地址 */
        ch395_set_socket_desip(ch395_socket->socket_index,
                                ch395_socket[ch395_socket->socket_index].des_ip);
        /* 设置 socket 0 协议类型 */
        ch395_set_socket_prot_type(ch395_socket->socket_index,
                                    PROTO_TYPE_UDP);

        /* 设置 socket 0 目的端口 */
        ch395_set_socket_desport(ch395_socket->socket_index,
                                   ch395_socket[ch395_socket->socket_index].des_port);
        /* 设置 socket 0 源端口 */
        ch395_set_socket_sourport(ch395_socket->socket_index,
                                   ch395_socket[ch395_socket->socket_index].sour_port);
        /* 检查是否成功 */
        g_ch395q_sta.ch395_error
        (ch395_open_socket(ch395_socket->socket_index));
        break;

    case CH395Q_SOCKET_TCP_CLIENT:
        /* socket 为 TCPClient 模式 */
        ch395_keeplive_set(); /* 保活设置 */
        /* 设置 socket 0 目标 IP 地址 */
        ch395_set_socket_desip(ch395_socket->socket_index,
                                ch395_socket[ch395_socket->socket_index].des_ip);
        /* 设置 socket 0 协议类型 */
        ch395_set_socket_prot_type(ch395_socket->socket_index,
                                    PROTO_TYPE_TCP);

        /* 设置 socket 0 目的端口 */
        ch395_set_socket_desport(ch395_socket->socket_index,
                                   ch395_socket[ch395_socket->socket_index].des_port);
        /* 设置 socket 0 源端口 */
        ch395_set_socket_sourport(ch395_socket->socket_index,
                                   ch395_socket[ch395_socket->socket_index].sour_port);
        /* 检查 socket 是否打开成功 */
        g_ch395q_sta.ch395_error
        (ch395_open_socket(ch395_socket->socket_index));
        /* 检查 tcp 连接是否成功 */
        g_ch395q_sta.ch395_error
        (ch395_tcp_connect(ch395_socket->socket_index));
        break;

    case CH395Q_SOCKET_TCP_SERVER:

```

```

/* socket 为TCPServer 模式 */
/* 设置 socket 0 目标 IP 地址 */
ch395_set_socket_desip(ch395_solect->socket_index,
                       ch395_solect[ch395_solect->socket_index].des_ip);
/* 设置 socket 0 协议类型 */
ch395_set_socket_prot_type(ch395_solect->socket_index,
                           PROTO_TYPE_TCP);

/* 设置 socket 0 源端口 */
ch395_set_socket_sourport(ch395_solect->socket_index,
                          ch395_solect[ch395_solect->socket_index].sour_port);

/* 检查 solect 是否打开成功 */
g_ch395q_sta.ch395_error
(ch395_open_socket(ch395_solect->socket_index));
/* 监听 tcp 连接 */
g_ch395q_sta.ch395_error
(ch395_tcp_listen(ch395_solect->socket_index));
break;
}

return 1;
}

```

通过传入的控制块判断协议的类型，程序根据这个协议的类型执行相应的代码。

### (7) 检测错误函数

```

/**
 * @brief      调试使用，显示错误代码，并停机
 * @param      ierror 检测命令
 * @retval     无
 */
void ch395_error(uint8_t ierror)
{
    if (ierror == CMD_ERR_SUCCESS)
    {
        return;          /* 操作成功 */
    }

#ifdef CH395_DEBUG
    printf("Error: %02X\r\n", (uint16_t)ierror);    /* 显示错误 */
#endif

    while ( 1 )
    {
        delay_ms(200);
        delay_ms(200);
    }
}

```

```

}
}

```

这个函数没有什么好讲解的，主要用来判断传入的形参是否为 CMD\_ERR\_SUCCESS，若传入的形参不为 CMD\_ERR\_SUCCESS，则程序会在 while 语句内运行。

### (8) 检测 PHY 状态函数

```

/**
 * @brief      CH395 PHY 状态
 * @param      phy_status: PHY 状态值
 * @retval     无
 */
void ch395_phy_status(uint8_t phy_status)
{
    switch (phy_status)
    {
        case PHY_DISCONN:
            printf("PHY DISCONN\r\n");
            break;
        case PHY_10M_FLL:
            printf("PHY 10M_FLL\r\n");
            break;
        case PHY_10M_HALF:
            printf("PHY 10M_HALF\r\n");
            break;
        case PHY_100M_FLL:
            printf("PHY 100M_FLL\r\n");
            break;
        case PHY_100M_HALF:
            printf("PHY 100M_HALF\r\n");
            break;
        default:
            printf("PHY AUTO\r\n");
            break;
    }
}

```

此函数根据传入的形参判断 PHY 处于那种状态，并以串口输出。

### (9) 设置 TCPClient 保活定时参数

```

/**
 * @brief      ch395_tcp 初始化
 * @param      无
 * @retval     无
 */
void ch395_hardware_init(void)
{

```

```
uint8_t i;
ch395_gpio_init();
spi1_init();

g_ch395q_sta.ch395_error = ch395_error;
g_ch395q_sta.ch395_phy_cb = ch395_phy_status;
g_ch395q_sta.ch395_reconnection = ch395_reconnection;
g_ch395q_sta.dhcp_status = DHCP_STA;

i = ch395_cmd_check_exist(0x65); /* 测试命令，用于测试硬件以及接口通讯 */

if (i != 0x9a)
{
    g_ch395q_sta.ch395_error(i); /* ch395q 检测错误 */
}

ch395_cmd_reset(); /* 对 ch395q 复位 */
delay_ms(100); /* 这里必须等待 100 以上延时 */

g_ch395q_sta.ch395_error(ch395_cmd_init()); /* 初始化 ch395q 命令 */

do
{
    g_ch395q_sta.phy_status = ch395_cmd_get_phy_status(); /* 获取 PHY 状态 */
    g_ch395q_sta.ch395_phy_cb(g_ch395q_sta.phy_status); /* 判断双工和网速模式 */
}
while(g_ch395q_sta.phy_status == PHY_DISCONN);

g_ch395q_sta.version = ch395_cmd_get_ver(); /* 获取版本 */
printf("CH395VER : %2x\r\n", (uint16_t)g_ch395q_sta.version);

i = ch395_dhcp_enable(1); /* 开启 DHCP */
g_ch395q_sta.ch395_error(i); /* ch395q 检测错误 */

delay_ms(1000); /* ch395q 初始化延时 */
}
```

此函数对 CH395Q 以太网芯片初始化和相关测试，测试完成之后开启 DHCP。

### (10) CH395Q 全局中断

```
/**
 * @brief      CH395 socket 中断, 在全局中断中被调用
 * @param      sockindex (0~7)
 * @retval     无
 */
```

```
void ch395_socket_interrupt(uint8_t sockindex)
{
    uint8_t sock_int_socket;
    uint16_t rx_len = 0;
    /* 获取 socket 的中断状态 */
    sock_int_socket = ch395_get_socket_int(sockindex);
    /* 发送缓冲区空闲，可以继续写入要发送的数据 */
    if (sock_int_socket & SINT_STAT_SENBUF_FREE)
    {

    }

    if (sock_int_socket & SINT_STAT_SEND_OK) /* 发送完成中断 */
    {

    }

    if (sock_int_socket & SINT_STAT_RECV) /* 接收中断 */
    {
        /* 获取当前缓冲区内数据长度 */
        g_ch395q_sta.socket[sockindex].config.recv.size
            = ch395_get_recv_length(sockindex);
        rx_len = g_ch395q_sta.socket[sockindex].config.recv.size;
        /* 读取数据 */
        ch395_get_recv_data(sockindex, rx_len,
            g_ch395q_sta.socket[sockindex].config.recv.buf);
        g_ch395q_sta.socket[sockindex].config.recv.buf[rx_len] = '\0';
        printf("%s", g_ch395q_sta.socket[sockindex].config.recv.buf);
    }

    if (sock_int_socket & SINT_STAT_CONNECT) /* 连接中断，仅在 TCP 模式下有效 */
    {
        if (g_ch395q_sta.socket[sockindex].config.proto
            == CH395Q_SOCKET_TCP_CLIENT)
        {
            ch395_set_keeplive(sockindex, 1); /* 打开 KEEPALIVE 保活定时器 */
            ch395_setttl_num(sockindex, 60); /* 设置 TTL */
        }
    }

    if (sock_int_socket & SINT_STAT_DISCONNECT) /* 断开中断，仅在 TCP 模式下有效 */
    {
        g_ch395q_sta.ch395_error(ch395_open_socket

```



```

        (g_ch395q_sta.socket[sockindex].config.socket_index));

switch(g_ch395q_sta.socket[sockindex].config.proto)
{
    case CH395Q_SOCKET_TCP_CLIENT:
        g_ch395q_sta.ch395_error(ch395_tcp_connect
            (g_ch395q_sta.socket[sockindex].config.socket_index));
        break;
    case CH395Q_SOCKET_TCP_SERVER:
        g_ch395q_sta.ch395_error(ch395_tcp_listen
            (g_ch395q_sta.socket[sockindex].config.socket_index));
        break;
    default:
        break;
}

delay_ms(200); /* 延时 200MS 后再次重试，没有必要过于频繁连接 */
}

if (sock_int_socket & SINT_STAT_TIM_OUT) /* 超时中断，仅在 TCP 模式下有效 */
{
    if (g_ch395q_sta.socket[sockindex].config.proto
        == CH395Q_SOCKET_TCP_CLIENT)
    {
        delay_ms(200); /* 延时 200MS 后再次重试，没有必要过于频繁连接 */
        g_ch395q_sta.ch395_error(ch395_open_socket
            (g_ch395q_sta.socket[sockindex].config.socket_index));
        g_ch395q_sta.ch395_error(ch395_tcp_connect
            (g_ch395q_sta.socket[sockindex].config.socket_index));
    }
}
}
}

```

根据 sockindex 形参的数值来选择 socket 通道，本教程的例程使用的是 socket0 通道，所以该变量的数值为 0。我们通过函数 ch395\_get\_socket\_int 获取 PHY 当前状态，并且根据状态执行相应的代码段。

### (11) socket 中断

```

/**
 * @brief      CH395 全局中断函数
 * @param      无
 * @retval     无
 */
void ch395_interrupt_handler(void)
{

```

```
uint16_t init_status;
uint8_t i;

init_status = ch395_cmd_get_glob_int_status_all();
/* 不可达中断, 读取不可达信息 */
if (init_status & GINT_STAT_UNREACH)
{
    ch395_cmd_get_unreachippt(g_ch395q_sta.ipinf_buf);
}
/* 产生 IP 冲突中断, 建议重新修改 CH395 的 IP, 并初始化 CH395 */
if (init_status & GINT_STAT_IP_CONFLI)
{
}

/* 产生 PHY 改变中断 */
if (init_status & GINT_STAT_PHY_CHANGE)
{
    g_ch395q_sta.phy_status = ch395_cmd_get_phy_status(); /* 获取 PHY 状态 */
}
/* 处理 DHCP 中断 */
if (init_status & GINT_STAT_DHCP)
{
    i = ch395_get_dhcp_status();

    switch (i)
    {
        case DHCP_UP:
            ch395_get_ipinf(g_ch395q_sta.ipinf_buf);
            printf("IP:%02d.%02d.%02d.%02d\r\n",
                (uint16_t)g_ch395q_sta.ipinf_buf[0],
                (uint16_t)g_ch395q_sta.ipinf_buf[1],
                (uint16_t)g_ch395q_sta.ipinf_buf[2],
                (uint16_t)g_ch395q_sta.ipinf_buf[3]);
            printf("GWIP:%02d.%02d.%02d.%02d\r\n",
                (uint16_t)g_ch395q_sta.ipinf_buf[4],
                (uint16_t)g_ch395q_sta.ipinf_buf[5],
                (uint16_t)g_ch395q_sta.ipinf_buf[6],
                (uint16_t)g_ch395q_sta.ipinf_buf[7]);
            printf("Mask:%02d.%02d.%02d.%02d\r\n",
                (uint16_t)g_ch395q_sta.ipinf_buf[8],
                (uint16_t)g_ch395q_sta.ipinf_buf[9],
```

```

        (uint16_t)g_ch395q_sta.ipinf_buf[10],
        (uint16_t)g_ch395q_sta.ipinf_buf[11]);
    printf("DNS1:%02d.%02d.%02d.%02d\r\n",
        (uint16_t)g_ch395q_sta.ipinf_buf[12],
        (uint16_t)g_ch395q_sta.ipinf_buf[13],
        (uint16_t)g_ch395q_sta.ipinf_buf[14],
        (uint16_t)g_ch395q_sta.ipinf_buf[15]);
    printf("DNS2:%02d.%02d.%02d.%02d\r\n",
        (uint16_t)g_ch395q_sta.ipinf_buf[16],
        (uint16_t)g_ch395q_sta.ipinf_buf[17],
        (uint16_t)g_ch395q_sta.ipinf_buf[18],
        (uint16_t)g_ch395q_sta.ipinf_buf[19]);
    g_ch395q_sta.dhcp_status = DHCP_UP;
    break;
default:
    g_ch395q_sta.dhcp_status = DHCP_DOWN;
    /* 设置默认 IP 地址信息 */
    printf("静态 IP 信息.....\r\n");
    break;
}
}

if (init_status & GINT_STAT_SOCK0)
{
    ch395_socket_interrupt(CH395Q_SOCKET_0);    /* 处理 socket 0 中断 */
}

if (init_status & GINT_STAT_SOCK1)
{
    ch395_socket_interrupt(CH395Q_SOCKET_1);    /* 处理 socket 1 中断 */
}

if (init_status & GINT_STAT_SOCK2)
{
    ch395_socket_interrupt(CH395Q_SOCKET_2);    /* 处理 socket 2 中断 */
}

if (init_status & GINT_STAT_SOCK3)
{
    ch395_socket_interrupt(CH395Q_SOCKET_3);    /* 处理 socket 3 中断 */
}

if (init_status & GINT_STAT_SOCK4)

```

```

{
    ch395_socket_interrupt(CH395Q_SOCKET_4);    /* 处理 socket 4 中断 */
}

if (init_status & GINT_STAT_SOCK5)
{
    ch395_socket_interrupt(CH395Q_SOCKET_5);    /* 处理 socket 5 中断 */
}

if (init_status & GINT_STAT_SOCK6)
{
    ch395_socket_interrupt(CH395Q_SOCKET_6);    /* 处理 socket 6 中断 */
}

if (init_status & GINT_STAT_SOCK7)
{
    ch395_socket_interrupt(CH395Q_SOCKET_7);    /* 处理 socket 7 中断 */
}
}

```

此函数是根据 INT#引脚的电平而调用，如果 INT#引脚的电平为低电平，则 CH395Q 触发了一个中断。它通过 ch395\_cmd\_get\_glob\_int\_status\_all 函数获取全局中断状态，根据这个中断状态执行相应的功能。

### (12) CH395 全局管理函数

```

/**
 * @brief      CH395 全局管理函数
 * @param      无
 * @retval     无
 */
void ch395q_handler(void)
{
    if (ch395_int_pin_wire == 0)
    {
        ch395_interrupt_handler();    /* 中断处理函数 */
    }

    g_ch395q_sta.ch395_reconnection();    /* 检测 PHY 状态，并重新连接 */
}

```

可以看到，此函数先判断 INT#的电平，如果为低电平，则调用 ch395\_interrupt\_handler 函数处理，接着调用 g\_ch395q\_sta.ch395\_reconnection 函数检测 PHY 状态，若 PHY 状态为断开，则关闭 socket 通道，直到网线重新插入才打开 socket 通道，同时触发重连函数。

### (13) 重接机制

```

/**
 * @brief      检测 PHY 状态，并重新连接

```

```

* @param      无
* @retval     无
*/
void ch395_reconnection(void)
{
    for (uint8_t socket_index = CH395Q_SOCKET_0 ;
        socket_index < CH395Q_SOCKET_7 ; socket_index ++ )
    {
        if (g_ch395q_sta.phy_status == PHY_DISCONN &&
            (g_ch395q_sta.dhcp_status == DHCP_UP
            || g_ch395q_sta.dhcp_status == DHCP_DOWN))
        {
            if (g_ch395q_sta.socket[socket_index].config.socket_enable
                == CH395Q_ENABLE)
            {
                ch395_close_socket(
                    g_ch395q_sta.socket[socket_index].config.socket_index);
                /* ch395q 检测错误 */
                g_ch395q_sta.ch395_error(ch395_dhcp_enable(0));
                g_ch395q_sta.socket[socket_index].config.socket_enable
                    = CH395Q_DISABLE;
                g_ch395q_sta.dhcp_status = DHCP_STA;
            }
        }
        else
        {
            if (g_ch395q_sta.phy_status != PHY_DISCONN &&
                g_ch395q_sta.socket[socket_index].config.socket_enable
                    == CH395Q_DISABLE)
            {
                ch395_cmd_reset(); /* 对 ch395q 复位 */
                delay_ms(100); /* 这里必须等待 100 以上延时 */
                g_ch395q_sta.ch395_error(ch395_cmd_init()); /* 初始化 ch395q 命令 */
                /* 开启 DHCP */
                g_ch395q_sta.ch395_error(ch395_dhcp_enable(1));

                do
                {
                    if (ch395_int_pin_wire == 0)
                    {
                        ch395_interrupt_handler(); /* 中断处理函数 */
                    }
                }
            }
        }
    }
}

```

```

while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获取 DHCP */

switch(g_ch395q_sta.socket[socket_index].config.proto)
{
    case CH395Q_SOCKET_UDP:
        /* socket 为 UDP 模式 */
        /* 设置 socket 0 目标 IP 地址 */
        ch395_set_socket_desip(socket_index,
                                g_ch395q_sta.socket[socket_index].config.des_ip);
        /* 设置 socket 0 协议类型 */
        ch395_set_socket_prot_type(socket_index,  PROTO_TYPE_UDP);
        /* 设置 socket 0 目的端口 */
        ch395_set_socket_desport(socket_index,
                                   g_ch395q_sta.socket[socket_index].config.des_port);
        /* 设置 socket 0 源端口 */
        ch395_set_socket_sourport(socket_index,
                                   g_ch395q_sta.socket[socket_index].config.sour_port);
        /* 检查是否成功 */
        g_ch395q_sta.ch395_error(ch395_open_socket(socket_index));
        break;
    case CH395Q_SOCKET_TCP_CLIENT:
        /* socket 为 TCPClient 模式 */
        ch395_keepalive_set(); /* 保活设置 */
        /* 设置 socket 0 目标 IP 地址 */
        ch395_set_socket_desip(socket_index,
                                g_ch395q_sta.socket[socket_index].config.des_ip);
        /* 设置 socket 0 协议类型 */
        ch395_set_socket_prot_type(socket_index,  PROTO_TYPE_TCP);
        /* 设置 socket 0 目的端口 */
        ch395_set_socket_desport(socket_index,
                                   g_ch395q_sta.socket[socket_index].config.des_port);
        /* 设置 socket 0 源端口 */
        ch395_set_socket_sourport(socket_index,
                                   g_ch395q_sta.socket[socket_index].config.sour_port);
        /* 检查 socket 是否打开成功 */
        g_ch395q_sta.ch395_error(ch395_open_socket(socket_index));
        /* 检查 tcp 连接是否成功 */
        g_ch395q_sta.ch395_error(ch395_tcp_connect(socket_index));
        break;
    case CH395Q_SOCKET_TCP_SERVER:
        /* socket 为 TCPServer 模式 */
        /* 设置 socket 0 目标 IP 地址 */
        ch395_set_socket_desip(socket_index,

```



```

        g_ch395q_sta.socket[socket_index].config.des_ip);
    /* 设置 socket 0 协议类型 */
    ch395_set_socket_prot_type(socket_index,  PROTO_TYPE_TCP);
    /* 设置 socket 0 源端口 */
    ch395_set_socket_sourport(socket_index,
        g_ch395q_sta.socket[socket_index].config.sour_port);
    /* 检查 socket 是否打开成功 */
    g_ch395q_sta.ch395_error(ch395_open_socket(socket_index));
    /* 监听 tcp 连接 */
    g_ch395q_sta.ch395_error(ch395_tcp_listen(socket_index));
    break;
}

g_ch395q_sta.socket[0].config.socket_enable = CH395Q_ENABLE;
}

}

}
}

```

此函数主要判断 PHY 状态，如果网线断开，则等待网线重新插入那一刻触发重连机制。

## 5. 文件 ch395.h

```

#ifndef __CH395_H
#define __CH395_H
#include "../BSP/CH395Q/ch395inc.h"
#include "../SYSTEM/sys/sys.h"
#include "../SYSTEM/usart/usart.h"
#include "../BSP/CH395Q/ch395inc.h"
#include "../BSP/CH395Q/ch395cmd.h"
#include "../SYSTEM/delay/delay.h"
#include "string.h"
#include "stdio.h"

/*****
/* 引脚 定义 */
#define CH395_SCS_GPIO_PORT          GPIOG
#define CH395_SCS_GPIO_PIN           GPIO_PIN_9
#define CH395_SCS_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOG_CLK_ENABLE(); }while(0)          /* PG 口时钟使能 */

#define CH395_INT_GPIO_PORT          GPIOG
#define CH395_INT_GPIO_PIN           GPIO_PIN_6
#define CH395_INT_GPIO_CLK_ENABLE()
do{ __HAL_RCC_GPIOG_CLK_ENABLE(); }while(0)          /* PG 口时钟使能 */

```

```

#define CH395_RST_GPIO_PORT          GPIOD
#define CH395_RST_GPIO_PIN           GPIO_PIN_7
#define CH395_RST_GPIO_CLK_ENABLE()

do{ __HAL_RCC_GPIOD_CLK_ENABLE(); }while(0)                /* PD 口时钟使能 */

/*****
/* SPI 片选引脚输出低电平 */
#define ch395_scs_low      HAL_GPIO_WritePin(GPIOG, GPIO_PIN_9, GPIO_PIN_RESET)
/* SPI 片选引脚输出高电平 */
#define ch395_scs_high     HAL_GPIO_WritePin(GPIOG, GPIO_PIN_9, GPIO_PIN_SET)
/* 获取 CH395 的 SPI 数据输出引脚电平 */
#define ch395_sdo_pin      HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_6)
/* 假定 CH395 的 INT#引脚,如果未连接那么也可以通过查询兼做中断输出的 SDO 引脚状态实现 */
#define ch395_int_pin_wire HAL_GPIO_ReadPin(GPIOG,GPIO_PIN_6)

typedef struct ch395q_socket_t
{
    uint8_t socket_enable;          /* Socket 使能 */
    uint8_t socket_index;           /* Socket 标号 */
    uint8_t proto;                  /* Socket 协议 */
    uint8_t des_ip[4];              /* 目的 IP 地址 */
    uint16_t des_port;              /* 目的端口 */
    uint16_t sour_port;             /* 源端口 */

    struct
    {
        uint8_t *buf;               /* 缓冲空间 */
        uint32_t size;              /* 缓冲空间大小 */
    } send;                         /* 发送缓冲 */

    struct
    {
        uint8_t *buf;               /* 缓冲空间 */
        uint32_t size;              /* 缓冲空间大小 */
    } recv;                         /* 接收缓冲 */

    struct
    {
        uint8_t ip[4];              /* IP 地址 */
        uint8_t gwip[4];            /* 网关 IP 地址 */
        uint8_t mask[4];            /* 子网掩码 */
        uint8_t dns1[4];            /* DNS 服务器 1 地址 */
    }

```

```

        uint8_t dns2[4];                /* DNS 服务器 2 地址 */
    } net_info;                          /* 网络信息 */

    struct
    {
        uint8_t ipaddr[4];              /* IP 地址 32bit*/
        uint8_t gwipaddr[4];            /* 网关地址 32bit*/
        uint8_t maskaddr[4];            /* 子网掩码 32bit*/
        uint8_t macaddr[6];             /* MAC 地址 48bit*/
    } net_config;                        /* 网络配置信息 */

} ch395_socket;

/* DHCP 状态 */
enum DHCP
{
    DHCP_UP = 0,                        /* DHCP 获取成功状态 */
    DHCP_DOWN,                          /* DHCP 获取失败状态 */
    DHCP_STA,                           /* DHCP 开启状态 */
};

struct ch395q_t
{
    uint8_t version;                    /* 版本信息 */
    uint8_t phy_status;                  /* PHY 状态 */
    uint8_t dhcp_status;                 /* DHCP 状态 */
    uint8_t ipinf_buf[20];               /* 获取 IP 信息 */

    struct
    {
        ch395_socket config;            /* 配置信息 */
    } socket[8];                        /* Socket 状态 */

    void (*ch395_error)(uint8_t i);      /* ch395q 错误检测函数 */
    void (*ch395_phy_cb)(uint8_t phy_status); /* ch395q phy 状态回调函数 */
    void (*ch395_reconnection)(void);    /* ch395q 重新连接函数 */
};

extern struct ch395q_t g_ch395q_sta;

/* CH395Q 模块 Socket 标号定义 */
#define CH395Q_SOCKET_0                0        /* Socket 0 */
#define CH395Q_SOCKET_1                1        /* Socket 1 */

```

```

#define CH395Q_SOCKET_2      2      /* Socket 2 */
#define CH395Q_SOCKET_3      3      /* Socket 3 */
#define CH395Q_SOCKET_4      4      /* Socket 4 */
#define CH395Q_SOCKET_5      5      /* Socket 5 */
#define CH395Q_SOCKET_6      6      /* Socket 6 */
#define CH395Q_SOCKET_7      7      /* Socket 7 */

/* 使能定义 */
#define CH395Q_DISABLE        1      /* 禁用 */
#define CH395Q_ENABLE        2      /* 使能 */

/* CH395Q 模块 Socket 协议类型定义 */
#define CH395Q_SOCKET_UDP      0      /* UDP */
#define CH395Q_SOCKET_TCP_CLIENT  1      /* TCP 客户端 */
#define CH395Q_SOCKET_TCP_SERVER  2      /* TCP 服务器 */

#define DEF_KEEP_LIVE_IDLE      (15*1000) /* 空闲时间 */
/* 间隔为 15 秒，发送一次 KEEPLIVE 数据包 */
#define DEF_KEEP_LIVE_PERIOD      (15*1000)
#define DEF_KEEP_LIVE_CNT        200

uint8_t ch395_read_data(void) ;
void ch395_write_cmd( uint8_t mcmd );
void ch395_write_data( uint8_t mdata );
void ch395q_handler(void);
void ch395_interrupt_handler(void);
void ch395_hardware_init(void);
uint8_t ch395q_socket_config(ch395_socket * ch395_socect);
void ch395_reconnection(void);

#endif
    
```

此函数主要声明 ch395.c 中的函数和结构体，这些函数和结构体可在其他文件中调用。至此，ch395.c/h 文件介绍完成。

## 1.5 移植 CH395Q 源码

在移植之前我们需要一个基础工程，这里我们使用裸机例程中的内存管理实验作为基础工程，我们在这个工程的基础上完成本章的 CH395Q 移植。

首先我们把内存管理实验重命名为“网络实验 1 CH395\_DHCP 实验”工程，然后在该工程的 Drivers\BSP 文件夹下创建 CH395Q 文件夹，并在此文件夹添加 ch395.c/h、ch395cmd.c/h 和 ch395inc.h 文件，这些文件我们可在正点原子移植好的例程中获取，如下图所示：

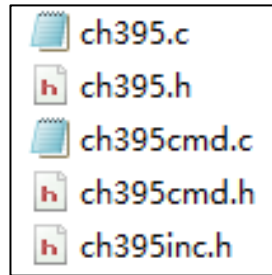


图 1.5.1 CH395Q 文件夹下的文件

可以看到,CH395Q 文件夹下保存着与 CH395Q 相关的源码,接着我们在工程的 Drivers/BSP 分组添加上图的.c 文件,如下图所示:

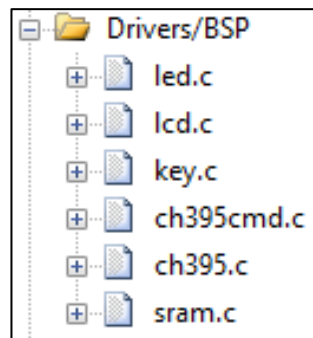


图 1.5.2 添加 CH395Q 驱动文件

前面笔者也讲到,本教程配套的例程使用的是 SPI 接口通讯,所以我们需要添加 SPI 驱动文件,这些文件可在战舰“实验 24 SPI 实验”下获取,并且把它复制到本实验的 Drivers/BSP 路径下,同时,在工程的 Drivers/BSP 分组上添加该驱动文件,如下图所示:

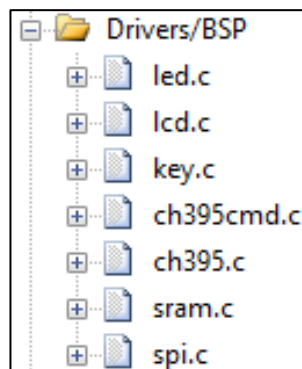


图 1.5.3 添加 SPI 驱动文件

值得注意的是,该驱动文件的 SPI 协议是使用 SPI2 接口实现的,而正点原子战舰开发板板载的 CH395Q 使用的是 SPI1 接口通讯,所以我们把该驱动接口修改为 SPI1 接口,修改过程很简单,这里笔者无需讲解了,大家请参考移植好的例程。

本教程是以标准例程-HAL 库版本的内存管理实验工程为基础工程,所以内存管理实验工程的工程名为“MALLOC”,为了规范工程,笔者建议将工程的目标名修改为“CH395Q”或根据读者的实际场景进行修改,修改如下图所示:



图 1.5.4 修改工程目标名称

到了这里，我们已经移植完成，编译工程，如果出现下图的错误，则添加 HAL 驱动文件。

```
Undefined symbol HAL_SPI_Init (referred from spi.o).
Undefined symbol HAL_SPI_TransmitReceive (referred from spi.o).
```

图 1.5.5 缺少 SPI 协议驱动

上图中的错误很容易解决，我们在工程的 Drivers/STM32F1xx\_HAL\_Driver 分组上添加 Drivers/STM32F1xx\_HAL\_Driver\Src 路径下的 stm32f1xx\_hal\_spi.c 文件即可解决。再一次编译，就不会出现错误和警告了。

为了工程整洁性，笔者把 Drivers\BSP 路径下未使用的驱动文件一并删除了，如下图所示：

修改前		修改后
<input type="checkbox"/> 24CXX	<input type="checkbox"/> LED	<input type="checkbox"/> BEEP
<input type="checkbox"/> ADC	<input type="checkbox"/> LSENS	<input type="checkbox"/> CH395Q
<input type="checkbox"/> BEEP	<input type="checkbox"/> NORFLASH	<input type="checkbox"/> KEY
<input type="checkbox"/> DAC	<input type="checkbox"/> NRF24L01	<input type="checkbox"/> LCD
<input type="checkbox"/> DHT11	<input type="checkbox"/> PWMDAC	<input type="checkbox"/> LED
<input type="checkbox"/> DMA	<input type="checkbox"/> REMOTE	<input type="checkbox"/> SPI
<input type="checkbox"/> DS18B20	<input type="checkbox"/> RS485	<input type="checkbox"/> SRAM
<input type="checkbox"/> IIC	<input type="checkbox"/> SPI	
<input type="checkbox"/> KEY	<input type="checkbox"/> SRAM	
<input type="checkbox"/> LCD	<input type="checkbox"/> STMFLASH	
	<input type="checkbox"/> TOUCH	

图 1.5.6 删除没有用到的驱动文件

为了代码整体简洁性，我们在工程上创建 ch395\_demo.c/h 文件，并把它们保存在 User\APP 路径下，同时，在工程的 User 分组下添加 ch395\_demo.c 文件。

至此，我们已经移植完毕，下面我们来测试一下网络是否正常。首先我们在 main.c 文件中编写测试代码，该测试代码如下所示：

```
#include "../SYSTEM/sys/sys.h"
#include "../SYSTEM/usart/usart.h"
#include "../SYSTEM/delay/delay.h"
#include "../BSP/LED/led.h"
#include "../BSP/LCD/lcd.h"
#include "../BSP/KEY/key.h"
#include "../MALLOC/malloc.h"
#include "../BSP/SRAM/sram.h"
#include "../BSP/CH395Q/ch395.h"
#include "../APP/ch395_demo.h"

int main(void)
{
    HAL_Init(); /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9); /* 设置时钟，72Mhz */
    delay_init(72); /* 延时初始化 */
```

```

    usart_init(115200);          /* 串口初始化为 115200 */
    led_init();                  /* 初始化 LED */
    lcd_init();                  /* 初始化 LCD */
    key_init();                  /* 初始化按键 */
    sram_init();                 /* 初始化外部 SRAM */
    my_mem_init(SRAMIN);         /* 初始化内部内存 */
    my_mem_init(SRAMEX);         /* 初始化外部内存 */
    ch395_hardware_init();       /* ch395 初始化 */

    ch395_demo();                /* 例程测试 */

}

```

main 函数调用了 ch395\_hardware\_init 函数初始化 CH395Q，接着调用 ch395\_demo 执行测试代码，该函数如下所示：

```

/**
 * @brief      显示实验信息
 * @param      无
 * @retval     无
 */
void ch395_show_mesg(void) {
    /* LCD 显示实验信息 */
    lcd_show_string(10, 10, 220, 32, 32, "STM32", RED);
    lcd_show_string(10, 47, 220, 24, 24, "CH395Q Client", RED);
    lcd_show_string(10, 76, 220, 16, 16, "ATOM@ALIEN TEK", RED);
    lcd_show_string(10, 97, 200, 16, 16, "KEY0: Send", BLUE);
    /* 串口输出实验信息 */
    printf("\n");
    printf("*****\r\n");
    printf("STM32\r\n");
    printf("CH395Q Client\r\n");
    printf("ATOM@ALIEN TEK\r\n");
    printf("KEY0: Send\r\n");
    printf("*****\r\n");
    printf("\r\n");
}

/**
 * @brief      例程测试
 * @param      无
 * @retval     无
 */
void ch395_demo(void) {
    ch395_show_mesg(); /* 显示信息 */
    do{

```

```

if (ch395_int_pin_wire == 0)
{
    ch395q_handler(); /* 中断处理函数 */
}
}

while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获取 DHCP*/
while(1){
    ch395q_handler();
}
}

```

可以看到，ch395\_demo 函数调用了 ch395\_show\_mesg 函数显示信息，程序继续往下执行，直到 dhcp\_status 不等于 DHCP\_STA 状态时，系统退出 do-while 循环，表示 DHCP 分配成功。

到了这里，我们测试一下网络是否正常，首先打开串口调式助手，并且把代码下载到开发板中，如下图所示：

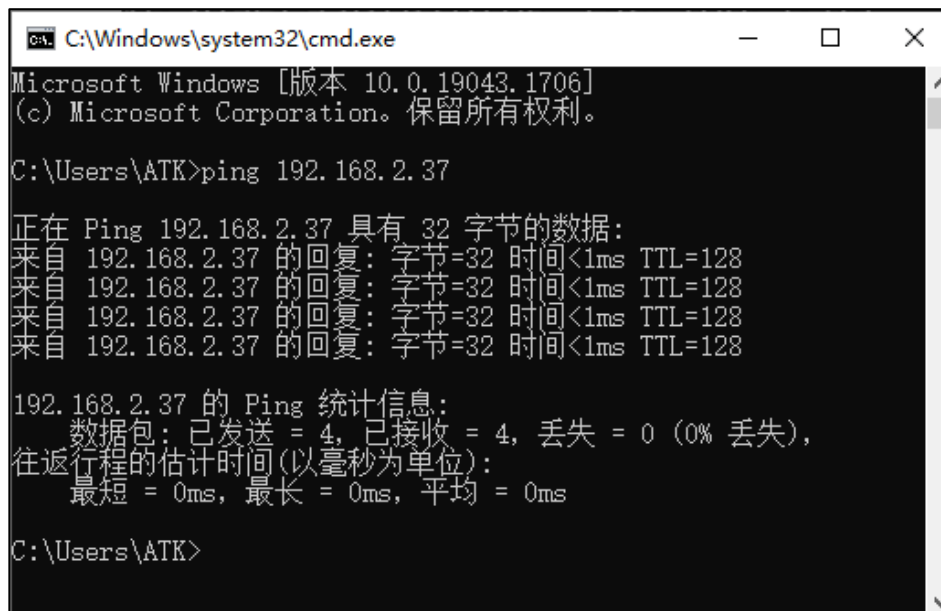
```

IP:192.168.02.37
GWIP:192.168.01.01
Mask:255.255.252.00
DNS1:202.96.128.86
DNS2:114.114.114.114

```

图 1.5.7 DHCP 分配 IP 地址

在 PC 机上按下“win+r”快捷键并输入 cmd 进去命令行，在该命令行上输入“ping 192.168.2.37”命令，如果命令行弹出“无法访问目标主机”字符串，则 PC 机无法与开发板通讯，显然我们移植的工程是失败的；如果命令行弹出“字节=32 时间<1ms TTL=255”字符串，则 PC 机能够与开发板通讯，证明该工程移植成功，下面我们在命令行 ping 一下图 1.3.7 的 IP 地址，如下图所示：



```

C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.19043.1706]
(c) Microsoft Corporation. 保留所有权利。

C:\Users\ATK>ping 192.168.2.37

正在 Ping 192.168.2.37 具有 32 字节的数据:
来自 192.168.2.37 的回复: 字节=32 时间<1ms TTL=128
来自 192.168.2.37 的回复: 字节=32 时间<1ms TTL=128
来自 192.168.2.37 的回复: 字节=32 时间<1ms TTL=128
来自 192.168.2.37 的回复: 字节=32 时间<1ms TTL=128

192.168.2.37 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 0ms, 最长 = 0ms, 平均 = 0ms

C:\Users\ATK>

```

图 1.5.8 ping IP 地址成功

该实验的实验工程，请参考《网络实验 1 CH395\_DHCP 实验》实验。



## 第二章 UDP 实验

UDP 协议在 TCP/IP 协议栈的传输层，它是无连接，也称透明连接，简单来讲：UDP 提供无连接的传输时，通信前不需要建立连接，一般用作于视频传输及音频传输的项目。这里笔者重点声明一下，本教程不会讲解各层协议的知识及原理，如读者有兴趣学习以太网的知识，请看正点原子的《lwIP 开发指南》，该教程已经很详细讲解了各层之间协议的知识 and 实现原理。

本章我们分为如下几个部分讲解：

### 2.1 UDP 配置流程

#### 2.2 UDP 实验

### 2.1 UDP 配置流程

CH395Q 以太网芯片实现 UDP 连接是非常简单的，我们只需要调用 ch595\_cmd.c 文件中的配置函数发送相应的命令即可完成，下面笔者将介绍 CH395Q 以太网芯片是如何实现 UDP 连接的，UDP 连接步骤如下所示：

第一步：将《网络实验 1 CH395\_DHCP 实验》实验拷贝并复制到该实验路径下，并把复制的工程命名为“网络实验 2 CH395\_UDP 实验”。

第二步：选择协议类型，这里分为三种协议模式，如下源码所示：

```
/* CH395Q 模块 Socket 协议类型定义 */
#define CH395Q_SOCKET_UDP          0                /* UDP */
#define CH395Q_SOCKET_TCP_CLIENT  1                /* TCP 客户端 */
#define CH395Q_SOCKET_TCP_SERVER  2                /* TCP 服务器 */
```

本章是 UDP 实验，所以选择 CH395Q\_SOCKET\_UDP 配置项。

第三步：设置源端口和目标端口。

第四步：ch395.c 文件下定义 IP 地址、子网掩码和网关等参数，这些内容笔者稍后在程序设计小节讲解。

至此，我们已经配置 UDP 完成，下面笔者将带领大家学习本章节实验的代码。

### 2.2 UDP 实验

#### 2.2.1 硬件设计

##### 1. 例程功能

在本实验中，开发板主控芯片通过 SPI 接口与 CH395Q 以太网芯片进行通讯，从而完成对 CH395Q 以太网芯片的功能配置、数据接收等功能，同时将 CH395Q 以太网芯片的 Socket0 配置为 UDP 模式，并可通过按键发送 UDP 广播数据至其他的 UDP 客户端，也能够接收其他 UDP 客户端广播的数据，并实时显示至串口调试助手。

该实验的实验工程，请参考《网络实验 2 CH395\_UDP 实验》。

## 2.2.2 程序设计

### 2.2.2.1 程序流程图

本实验的程序流程图，如下图所示：

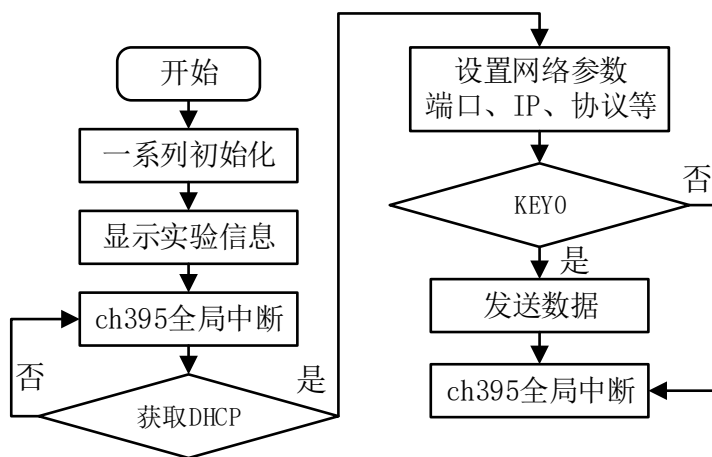


图 2.2.2.1.1 UDP 实验流程图

### 2.2.2.2 程序解析

关于 UDP 实验的程序解析，笔者着重讲解 ch395\_demo.c/h 文件，ch395\_demo.h 文件主要声明 ch395\_demo 函数提供外部文件使用，而 ch395\_demo.c 文件定义了三个函数，这些函数如下表所示：

函数	功能描述
ch395_show_mesg	显示实验信息
ch395_demo	例程测试

表 2.2.2.2.1 UDP 实验函数

下面是对本实验的 ch395\_demo.c 文件中的函数进行介绍，如下所示：

#### 1. 函数 ch395\_show\_mesg

该函数主要的作用是显示实验信息，如下源码所示：

```

/**
 * @brief      显示实验信息
 * @param      无
 * @retval     无
 */
void ch395_show_mesg(void)
{
    /* LCD 显示实验信息 */
    lcd_show_string(10, 10, 220, 32, 32, "STM32", RED);
    lcd_show_string(10, 47, 220, 24, 24, "CH395Q UDP", RED);
    lcd_show_string(10, 76, 220, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(10, 97, 200, 16, 16, "KEY0: Send", BLUE);
}

```

```

/* 串口输出实验信息 */
printf("\n");
printf("*****\r\n");
printf("STM32\r\n");
printf("CH395Q UDP\r\n");
printf("ATOM@ALIEN TEK\r\n");
printf("KEY0: Send\r\n");
printf("*****\r\n");
printf("\r\n");
}

```

此函数调用 lcd\_show\_string 和 printf 函数分别在 LCD 和串口上显示实验信息。

## 2. 函数 ch395\_demo

此函数主要设置 UDP 相关网络参数，例如配置协议、IP 及端口等信息，如下源码所示：

```

/* 本地网络信息：IP 地址、网关地址、子网掩码和 MAC 地址 */
uint8_t ch395_ipaddr[4]    = {192,168,1,10};
uint8_t ch395_gw_ipaddr[4] = {192,168,1,1};
uint8_t ch395_ipmask[4]    = {255,255,255,0};
uint8_t ch395_macaddr[6]   = {0xB8,0xAE,0x1D,0x00,0x00,0x00};

/* 远程 IP 地址设置 */
uint8_t ch395_des_ipaddr[4] = {192,168,1,111};
static uint8_t socket0_send_buf[] = {"This is from CH395Q\r\n"};
static uint8_t socket0_recv_buf[1024];

ch395_socket cha95_sockct_sta[8];

/**
 * @brief      例程测试
 * @param      无
 * @retval     无
 */
void ch395_demo(void)
{
    uint8_t key = 0;

    ch395_show_mesg(); /* 显示信息 */

    do
    {
        ch395q_handler();
    }

    while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获取 DHCP*/

    /* 使能 socket 接口 */

```

```
cha95_sockct_sta[0].socket_enable = CH395Q_ENABLE;

/* 设置 socket 接口 */
cha95_sockct_sta[0].socket_index = CH395Q_SOCKET_0;
/* 设置目标 IP 地址 */
memcpy(cha95_sockct_sta[0].des_ip, ch395_des_ipaddr,
        sizeof(cha95_sockct_sta[0].des_ip));
/* 设置静态本地 IP 地址 */
memcpy(cha95_sockct_sta[0].net_config.ipaddr, ch395_ipaddr,
        sizeof(cha95_sockct_sta[0].net_config.ipaddr));
/* 设置静态网关 IP 地址 */
memcpy(cha95_sockct_sta[0].net_config.gwipaddr, ch395_gw_ipaddr,
        sizeof(cha95_sockct_sta[0].net_config.gwipaddr));
/* 设置静态子网掩码地址 */
memcpy(cha95_sockct_sta[0].net_config.maskaddr, ch395_ipmask,
        sizeof(cha95_sockct_sta[0].net_config.maskaddr));
/* 设置静态 MAC 地址 */
memcpy(cha95_sockct_sta[0].net_config.macaddr, ch395_macaddr,
        sizeof(cha95_sockct_sta[0].net_config.macaddr));

/* 目标端口 */
cha95_sockct_sta[0].des_port = 8080;
/* 源端口 */
cha95_sockct_sta[0].sour_port = 8080;
/* 设置协议 */
cha95_sockct_sta[0].proto = CH395Q_SOCKET_UDP;
/* 发送数据 */
cha95_sockct_sta[0].send.buf = socket0_send_buf;
/* 发送数据大小 */
cha95_sockct_sta[0].send.size = sizeof(socket0_send_buf);
/* 接收数据缓冲区 */
cha95_sockct_sta[0].recv.buf = socket0_recv_buf;
/* 接收数据大小 */
cha95_sockct_sta[0].recv.size = sizeof(socket0_recv_buf);
/* 配置 socket 参数 */
ch395q_socket_config(&cha95_sockct_sta[0]);

while(1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)
    {
        ch395_send_data(0, (uint8_t *)socket0_send_buf,
                        strlen((char *)socket0_send_buf));
    }
}
```

```

    }

    ch395q_handler();
}
}

```

首先笔者声明了 IP 地址、子网掩码、网关和 MAC 地址，这些网络参数是为了 DHCP 失败时，可设置默认的网络信息，ch395\_des\_ipaddr 数组需要填写电脑的 IP 地址（必须填写正确），而 socket0\_send\_buf 和 socket0\_recv\_buf 分别为发送的数据和接收缓冲区，cha95\_socket\_sta 变量的类型为 ch395\_socket，该结构体描述每一个 socket 网络参数，该结构体如下所示：

```

typedef struct ch395q_socket_t
{
    uint8_t socket_enable;           /* Socket 使能 */
    uint8_t socket_index;           /* Socket 标号 */
    uint8_t proto;                  /* Socket 协议 */
    uint8_t des_ip[4];              /* 目的 IP 地址 */
    uint16_t des_port;              /* 目的端口 */
    uint16_t sour_port;            /* 源端口 */

    struct
    {
        uint8_t *buf;              /* 缓冲空间 */
        uint32_t size;             /* 缓冲空间大小 */
    } send;                        /* 发送缓冲 */

    struct
    {
        uint8_t *buf;              /* 缓冲空间 */
        uint32_t size;             /* 缓冲空间大小 */
    } recv;                        /* 接收缓冲 */

    struct
    {
        uint8_t ip[4];             /* IP 地址 */
        uint8_t gwip[4];           /* 网关 IP 地址 */
        uint8_t mask[4];           /* 子网掩码 */
        uint8_t dns1[4];           /* DNS 服务器 1 地址 */
        uint8_t dns2[4];           /* DNS 服务器 2 地址 */
    } net_info;                   /* 网络信息 */

    struct
    {
        uint8_t ipaddr[4];         /* IP 地址 32bit */
        uint8_t gwipaddr[4];      /* 网关地址 32bit */
    }

```

```

        uint8_t maskaddr[4];           /* 子网掩码 32bit*/
        uint8_t macaddr[6];           /* MAC 地址 48bit*/
    } net_config;                      /* 网络配置信息 */

} ch395_socket;
    
```

此结构体笔者分为四个部分讲解，如下：

#### ① 描述每一个 socket 共同部分

```

uint8_t socket_enable;                /* Socket 使能 */
uint8_t socket_index;                /* Socket 标号 */
uint8_t proto;                       /* Socket 协议 */
uint8_t des_ip[4];                   /* 目的 IP 地址 */
uint16_t des_port;                   /* 目的端口 */
uint16_t sour_port;                  /* 源端口 */
    
```

socket\_enable 为使能 socket，因为 CH395Q 可以开启 8 个 socket 接口，所以笔者使用这个成员变量描述 socket 接口是否打开；socket\_index 为 socket 的编号，例如：0~7 的 socket 接口；proto 为协议选择，描述这个 socket 接口使用那个协议，例如：UDP、TCPClient 和 TCPServer 协议类型；des\_ip[4] 设置目标 IP 地址，这个成员变量主要用作于 UDP 和 TCPClient 协议类型；des\_port 和 sour\_port 就是设置目标端口和源端口。

#### ② 收发结构体

```

struct
{
    uint8_t *buf;                     /* 缓冲空间 */
    uint32_t size;                    /* 缓冲空间大小 */
} send;                              /* 发送缓冲 */

struct
{
    uint8_t *buf;                     /* 缓冲空间 */
    uint32_t size;                    /* 缓冲空间大小 */
} recv;
    
```

每一个 socket 都可以收发数据，所以笔者在这里定义了收发结构体，用来描述发送缓冲区和发送缓冲区大小以及接收缓冲区和接收缓冲区大小。

#### ③ 网络信息（DHCP）

```

struct
{
    uint8_t ip[4];                    /* IP 地址 */
    uint8_t gwip[4];                 /* 网关 IP 地址 */
    uint8_t mask[4];                 /* 子网掩码 */
    uint8_t dns1[4];                 /* DNS 服务器 1 地址 */
    uint8_t dns2[4];                 /* DNS 服务器 2 地址 */
} net_info;                          /* 网络信息 */
    
```

该结构体的成员变量是用来保存 DHCP 分配的网络信息，例如：IP 地址、子网掩码、网关以及 DNS 服务器地址。

#### ④ 网络配置信息（静态）

```
struct
{
    uint8_t ipaddr[4];           /* IP 地址 32bit*/
    uint8_t gwipaddr[4];        /* 网关地址 32bit*/
    uint8_t maskaddr[4];        /* 子网掩码 32bit*/
    uint8_t macaddr[6];         /* MAC 地址 48bit*/
} net_config;                  /* 网络配置信息 */
```

此结构体就是保存静态的网络信息，例如：DHCP 不成功时，我们设置 CH395Q 的网络信息为静态网络信息。

至此，ch395\_socket 结构体讲解完毕，总的来说：它就是用来描述每一个 socket 接口的网络参数等信息。接着回到 ch395\_demo 函数讲解，该函数调用 ch395\_show\_mesg 函数显示实验信息，并使用 do-while 语句获取 DHCP 是否成功，如果获取成功，则 dhcp\_status 为 DHCP\_UP，否则为 DHCP\_DOWN，然后设置 socket 接口网络信息，例如：socket 编号、静态网络信息、端口号及协议类型等参数，最后等待连接，连接完成后按下 KEY0\_PRES 就可以发送数据了。

**注意：协议类型为 CH395Q\_SOCKET\_UDP。**

### 2.2.3 下载验证

下载代码完成后，打开网络调试助手，等待开发板的 LCD 出现下图所示的界面。



图 2.2.3.1 远端 IP 地址设置

在网络调试助手上点击“连接”按钮，如下图所示：

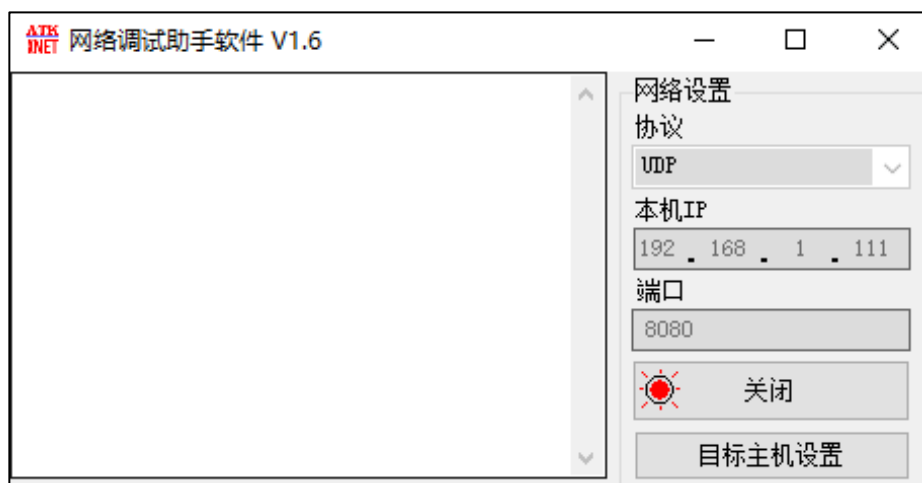


图 2.2.3.2 网络调试助手设置

设置完网络调试助手后在发送框填入要发送的数据，这里输入要发送的数据：ALIENTEK DATA，然后点击发送，这时串口调试助手显示接收的数据，接着通过按下 KEY0，向电脑端发

送数据 “This is from CH395Q\r\n” 如下图所示，表明网络调试助手接收到开发板发送的数据，这里我们按了 13 次 KEY0，因此在网络调试助手上有 13 行数据。

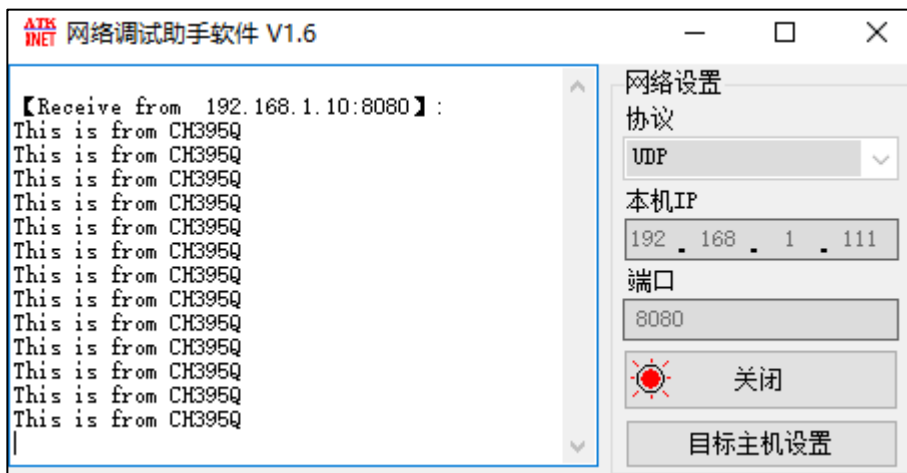


图 2.2.3.3 UDP 测试



## 第三章 TCP\_Client 实验

本章的目标是完成开发板和电脑之间的 TCP 通信，开发板作为客户端，网络调试助手作为 TCP 服务端，实验中我们通过电脑端的网络调试助手发送数据到开发板，开发板接收数据在串口上显示，同时开发板通过按键发送数据到网络调试助手。

本章我们分为如下几个部分讲解：

### 3.1 TCPClient 配置流程

#### 3.2 TCPClient 实验

### 3.1 TCPClient 配置流程

CH395Q 以太网芯片实现 TCPClient 连接是非常简单的，我们只需要调用 ch595\_cmd.c 文件中的配置函数发送相应的命令即可完成，下面笔者将介绍 CH395Q 以太网芯片是如何实现 TCPClient 连接的，如下步骤所示：

第一步：将《网络实验 2 CH395\_UDP 实验》实验拷贝并复制到该实验路径下，并把复制的工程命名为“网络实验 3 CH395\_TCP 客户端实验”。

第二步：选择协议类型，这里分为三种协议，如下源码所示：

```
/* CH395Q 模块 Socket 协议类型定义 */  
#define CH395Q_SOCKET_UDP          0          /* UDP */  
#define CH395Q_SOCKET_TCP_CLIENT  1          /* TCP 客户端 */  
#define CH395Q_SOCKET_TCP_SERVER  2          /* TCP 服务器 */
```

本章是 TCPClient 实验，所以选择 CH395Q\_SOCKET\_TCP\_CLIENT 配置项。

**注意：本实验和“网络实验 2 CH395\_UDP 实验”一样，仅修改显示信息和协议类型即可实现 TCPClient。**

第三步：设置源端口和目标端口。

第四步：ch395.c 文件下定义 IP 地址、子网掩码和网关等参数，这些内容笔者稍后在程序设计小节讲解。

至此，我们已经配置 TCPClient 完成，下面笔者将带领大家学习本章节实验的代码。

### 3.2 TCPClient 实验

#### 3.2.1 硬件设计

##### 1. 例程功能

在本实验中，开发板主控芯片通过 SPI 接口与 CH395Q 以太网芯片进行通讯，从而完成对 CH395Q 以太网芯片的功能配置、数据接收等功能，同时将 CH395Q 以太网芯片的 Socket0 配置为 TCP 客户端，并可通过按键向所连接的 TCP 服务器发送数据，也能够接收来自 TCP 服务器的数据，并实时显示至串口调试助手。

该实验的实验工程，请参考《网络实验 3 CH395\_TCP 客户端实验》。

## 3.2.2 程序设计

### 3.2.2.1 程序流程图

本实验的程序流程图，如下图所示：

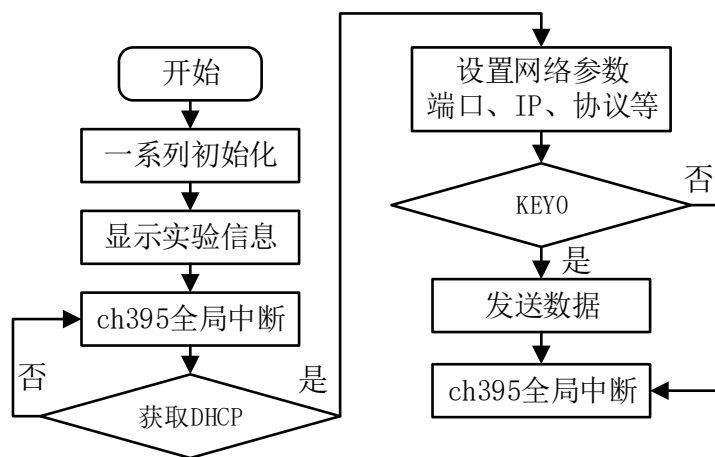


图 3.2.2.1.1 TCPClient 实验流程图

### 3.2.2.2 程序解析

关于 TCPClient 实验的程序解析，笔者着重讲解 ch395\_demo.c/h 文件，ch395\_demo.h 文件主要声明 ch395\_demo 函数提供外部文件使用，而 ch395\_demo.c 文件定义了三个函数，这些函数如下表所示：

函数	功能描述
ch395_show_mesg	显示实验信息
ch395_demo	例程测试

表 2.2.2.2.1 TCPClient 实验函数

下面是对本实验的 ch395\_demo.c 文件中的函数进行介绍，如下所示：

#### 1. 函数 ch395\_show\_mesg

该函数主要的作用是显示实验信息，如下源码所示：

```

/**
 * @brief      显示实验信息
 * @param      无
 * @retval     无
 */
void ch395_show_mesg(void)
{
    /* LCD 显示实验信息 */
    lcd_show_string(10, 10, 220, 32, 32, "STM32", RED);
    lcd_show_string(10, 47, 220, 24, 24, "CH395Q TCPClient", RED);
    lcd_show_string(10, 76, 220, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(10, 97, 200, 16, 16, "KEY0: Send", BLUE);
}
    
```

```

/* 串口输出实验信息 */
printf("\n");
printf("*****\r\n");
printf("STM32\r\n");
printf("CH395Q TCPClient\r\n");
printf("ATOM@ALIEN TEK\r\n");
printf("KEY0: Send\r\n");
printf("*****\r\n");
printf("\r\n");
}

```

此函数调用 lcd\_show\_string 和 printf 函数分别在 LCD 和串口上显示实验信息。

## 2. 函数 ch395\_demo

此函数主要设置 TCPClient 相关网络参数，如配置协议、IP 及端口等信息，如下源码所示：

```

/* 本地网络信息：IP 地址、网关地址、子网掩码和 MAC 地址 */
uint8_t ch395_ipaddr[4]    = {192,168,1,10};
uint8_t ch395_gw_ipaddr[4] = {192,168,1,1};
uint8_t ch395_ipmask[4]    = {255,255,255,0};
uint8_t ch395_macaddr[6]   = {0xB8,0xAE,0x1D,0x00,0x00,0x00};
/* 远程 IP 地址设置 */
uint8_t ch395_des_ipaddr[4] = {192,168,1,111};
static uint8_t socket0_send_buf[] = {"This is from CH395Q\r\n"};
static uint8_t socket0_recv_buf[1024];

ch395_socket cha95_sockct_sta[8];

/**
 * @brief      例程测试
 * @param      无
 * @retval     无
 */
void ch395_demo(void)
{
    uint8_t key = 0;

    ch395_show_mesg(); /* 显示信息 */

    do
    {
        ch395q_handler();
    }

    while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获取 DHCP */
    /* 使能 socket 接口 */
    cha95_sockct_sta[0].socket_enable = CH395Q_ENABLE;

```

```

/* 设置 socket 接口 */
cha95_sockct_sta[0].socket_index = CH395Q_SOCKET_0;
/* 设置目标 IP 地址 */
memcpy(cha95_sockct_sta[0].des_ip, ch395_des_ipaddr,
        sizeof(cha95_sockct_sta[0].des_ip));
/* 设置静态本地 IP 地址 */
memcpy(cha95_sockct_sta[0].net_config.ipaddr, ch395_ipaddr,
        sizeof(cha95_sockct_sta[0].net_config.ipaddr));
/* 设置静态网关 IP 地址 */
memcpy(cha95_sockct_sta[0].net_config.gwipaddr, ch395_gw_ipaddr,
        sizeof(cha95_sockct_sta[0].net_config.gwipaddr));
/* 设置静态子网掩码地址 */
memcpy(cha95_sockct_sta[0].net_config.maskaddr, ch395_ipmask,
        sizeof(cha95_sockct_sta[0].net_config.maskaddr));
/* 设置静态 MAC 地址 */
memcpy(cha95_sockct_sta[0].net_config.macaddr, ch395_macaddr,
        sizeof(cha95_sockct_sta[0].net_config.macaddr));
/* 目标端口 */
cha95_sockct_sta[0].des_port = 8080;
/* 源端口 */
cha95_sockct_sta[0].sour_port = 8080;
/* 设置协议 */
cha95_sockct_sta[0].proto = CH395Q_SOCKET_TCP_CLIENT;
/* 发送数据 */
cha95_sockct_sta[0].send.buf = socket0_send_buf;
/* 发送数据大小 */
cha95_sockct_sta[0].send.size = sizeof(socket0_send_buf);
/* 接收数据缓冲区 */
cha95_sockct_sta[0].recv.buf = socket0_recv_buf;
/* 接收数据大小 */
cha95_sockct_sta[0].recv.size = sizeof(socket0_recv_buf);
/* 配置 socket 参数 */
ch395q_socket_config(&cha95_sockct_sta[0]);

while(1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)
    {
        ch395_send_data(0, (uint8_t *)socket0_send_buf,
                        strlen((char *)socket0_send_buf));
    }
}

```

```

        ch395q_handler();
    }
}

```

首先笔者声明了 IP 地址、子网掩码、网关和 MAC 地址，这些网络参数是为了 DHCP 失败时，可设置默认的网络信息，ch395\_des\_ipaddr 数组需要填写电脑的 IP 地址（必须填写正确），而 socket0\_send\_buf 和 socket0\_recv\_buf 分别为发送的数据和接收缓冲区，cha95\_socket\_sta 变量的类型为 ch395\_socket，该结构体描述每一个 socket 网络参数，该结构体如下所示：

```

typedef struct ch395q_socket_t
{
    uint8_t socket_enable;           /* Socket 使能 */
    uint8_t socket_index;           /* Socket 标号 */
    uint8_t proto;                   /* Socket 协议 */
    uint8_t des_ip[4];               /* 目的 IP 地址 */
    uint16_t des_port;               /* 目的端口 */
    uint16_t sour_port;              /* 源端口 */

    struct
    {
        uint8_t *buf;                /* 缓冲空间 */
        uint32_t size;                /* 缓冲空间大小 */
    } send;                           /* 发送缓冲 */

    struct
    {
        uint8_t *buf;                /* 缓冲空间 */
        uint32_t size;                /* 缓冲空间大小 */
    } recv;                           /* 接收缓冲 */

    struct
    {
        uint8_t ip[4];                /* IP 地址 */
        uint8_t gwip[4];              /* 网关 IP 地址 */
        uint8_t mask[4];              /* 子网掩码 */
        uint8_t dns1[4];              /* DNS 服务器 1 地址 */
        uint8_t dns2[4];              /* DNS 服务器 2 地址 */
    } net_info;                       /* 网络信息 */

    struct
    {
        uint8_t ipaddr[4];            /* IP 地址 32bit*/
        uint8_t gwipaddr[4];          /* 网关地址 32bit*/
        uint8_t maskaddr[4];          /* 子网掩码 32bit*/
    }
}

```

```
uint8_t macaddr[6];          /* MAC 地址 48bit*/
} net_config;                /* 网络配置信息 */

} ch395_socket;
```

此结构体笔者分为四个部分讲解，如下：

#### ① 描述每一个 socket 共同部分

```
uint8_t socket_enable;      /* Socket 使能 */
uint8_t socket_index;      /* Socket 标号 */
uint8_t proto;              /* Socket 协议 */
uint8_t des_ip[4];          /* 目的 IP 地址 */
uint16_t des_port;          /* 目的端口 */
uint16_t sour_port;         /* 源端口 */
```

socket\_enable 为使能 socket，因为 CH395Q 可以开启 8 个 socket 接口，所以笔者使用这个成员变量描述 socket 接口是否打开；socket\_index 为 socket 的编号，例如：0~7 的 socket 接口；proto 为协议选择，描述这个 socket 接口使用那个协议，例如：UDP、TCPClient 和 TCPServer 协议类型；des\_ip[4]设置目标 IP 地址，这个成员变量主要用作于 UDP 和 TCPClient 协议类型；des\_port 和 sour\_port 就是设置目标端口和源端口。

#### ② 收发结构体

```
struct
{
    uint8_t *buf;            /* 缓冲空间 */
    uint32_t size;           /* 缓冲空间大小 */
} send;                      /* 发送缓冲 */

struct
{
    uint8_t *buf;            /* 缓冲空间 */
    uint32_t size;           /* 缓冲空间大小 */
} recv;
```

每一个 socket 都可以收发数据，所以笔者在这里定义了收发结构体，用来描述发送缓冲区和发送缓冲区大小以及接收缓冲区和接收缓冲区大小。

#### ③ 网络信息（DHCP）

```
struct
{
    uint8_t ip[4];           /* IP 地址 */
    uint8_t gwip[4];         /* 网关 IP 地址 */
    uint8_t mask[4];         /* 子网掩码 */
    uint8_t dns1[4];         /* DNS 服务器 1 地址 */
    uint8_t dns2[4];         /* DNS 服务器 2 地址 */
} net_info;                  /* 网络信息 */
```

该结构体的成员变量是用来保存 DHCP 分配的网络信息，例如：IP 地址、子网掩码、网关以及 DNS 服务器地址。

#### ④ 网络配置信息（静态）

```
struct
{
    uint8_t ipaddr[4];           /* IP 地址 32bit*/
    uint8_t gwipaddr[4];        /* 网关地址 32bit*/
    uint8_t maskaddr[4];        /* 子网掩码 32bit*/
    uint8_t macaddr[6];         /* MAC 地址 48bit*/
} net_config;                  /* 网络配置信息 */
```

此结构体就是保存静态的网络信息，例如：DHCP 不成功时，我们设置 CH395Q 的网络信息为静态网络信息。

至此，ch395\_socket 结构体讲解完毕，总的来说：它就是用来描述每一个 socket 接口的网络参数等信息。接着回到 ch395\_demo 函数讲解，该函数调用 ch395\_show\_mesg 函数显示实验信息，并使用 do while 语句获取 DHCP 是否成功，如果获取成功，则 dhcp\_status 为 DHCP\_UP，否则为 DHCP\_DOWN，然后设置 socket 接口网络信息，例如：socket 编号、静态网络信息、端口号及协议类型等参数，最后等待连接，连接完成后按下 KEY0\_PRES 就可以发送数据了。

**注意：协议类型为 CH395Q\_SOCKET\_TCP\_CLIENT。**

### 3.2.3 下载验证

下载代码完成后，打开网络调试助手，等待开发板的 LCD 出现下图所示的界面。



图 3.2.3.1 远端 IP 地址设置

在网络调试助手上点击“连接”按钮，如下图所示：

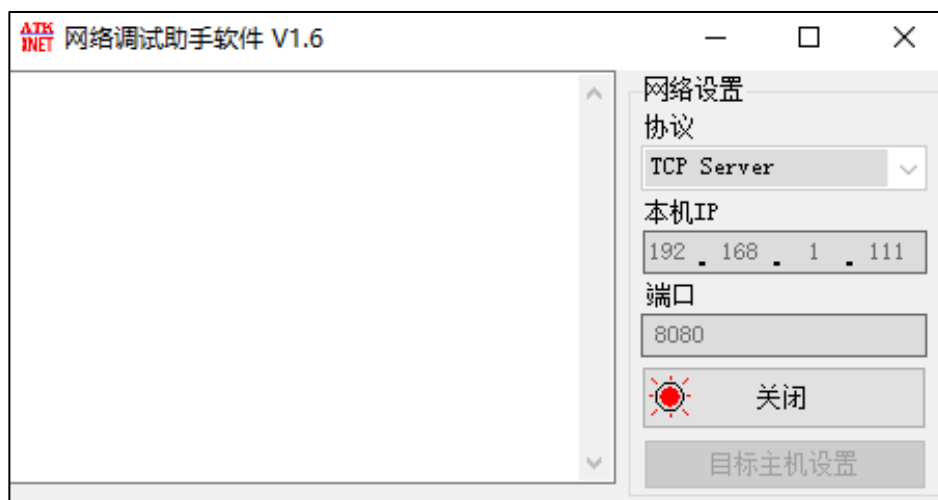


图 3.2.3.2 网络调试助手设置

设置完网络调试助手后在发送框填入要发送的数据，这里输入要发送的数据：ALIENTEK DATA，然后点击发送，这时串口调试助手显示接收的数据，接着通过按下 KEY0，向电脑端发送数据“This is from CH395Q\r\n”如下图所示，表明网络调试助手接收到开发板发送的数据，

这里我们按了 13 次 KEY0，因此在网络调试助手上有 13 行数据。

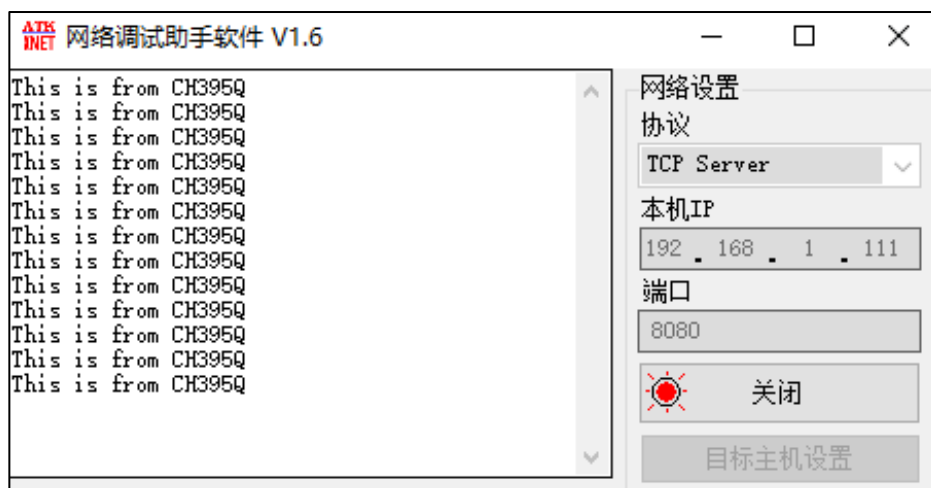


图 3.2.3.3 TCPClient 测试



## 第四章 TCP\_Server 实验

本章采用 TCP 协议完成开发板和电脑之间的 TCPServber 通信。在本章中开发板做 TCP 服务器，网络调试助手做 TCP 客户端，实验中我们通过电脑端的网络调试助手给开发板发送数据，开发板接收数据在串口上显示，同时也可以通过按键从开发板向网络调试助手发送数据。

本章我们分为如下几个部分讲解：

### 4.1 TCPServer 配置流程

### 4.2 TCPServer 实验

#### 4.1 TCPServer 配置流程

CH395Q 以太网芯片实现 TCPServer 连接是非常简单的，我们只需要调用 ch595\_cmd.c 文件中的配置函数发送相应的命令即可完成，下面笔者将介绍 CH395Q 以太网芯片是如何实现 TCPClient 连接的，如下步骤所示：

第一步：将《网络实验 2 CH395\_UDP 实验》实验拷贝并复制到该实验路径下，并把复制的工程命名为“网络实验 3 CH395\_TCP 服务器实验”。

第二步：选择协议类型，这里分为三种协议，如下源码所示：

```
/* CH395Q 模块 Socket 协议类型定义 */
#define CH395Q_SOCKET_UDP          0          /* UDP */
#define CH395Q_SOCKET_TCP_CLIENT  1          /* TCP 客户端 */
#define CH395Q_SOCKET_TCP_SERVER  2          /* TCP 服务器 */
```

本章是 TCPServer 实验，所以选择 CH395Q\_SOCKET\_TCP\_SERVER 配置项。

**注意：本实验和“网络实验 2 CH395\_UDP 实验”一样，仅修改显示信息和协议类型即可实现 TCPClient。**

第三步：设置源端口和目标端口。

第四步：ch395.c 文件下定义 IP 地址、子网掩码和网关等参数，这些内容笔者稍后在程序设计小节讲解。

至此，我们已经配置 TCPClient 完成，下面笔者将带领大家学习本章节实验的代码。

#### 4.2 TCPServer 实验

##### 4.2.1 硬件设计

##### 1. 例程功能

在本实验中，开发板主控芯片通过 SPI 接口与 CH395Q 以太网芯片进行通讯，从而完成对 CH395Q 以太网芯片的功能配置、数据接收等功能，同时将 CH395Q 以太网芯片的 Socket0 配置为 TCP 服务器，并可通过按键向连接的 TCP 客户端发送数据，也能够接收来自 TCP 客户端的数据，并实时显示至串口调试助手。

该实验的实验工程，请参考《网络实验 4 CH395\_TCP 服务器实验》。

## 4.2.2 程序设计

### 4.2.2.1 程序流程图

本实验的程序流程图，如下图所示：

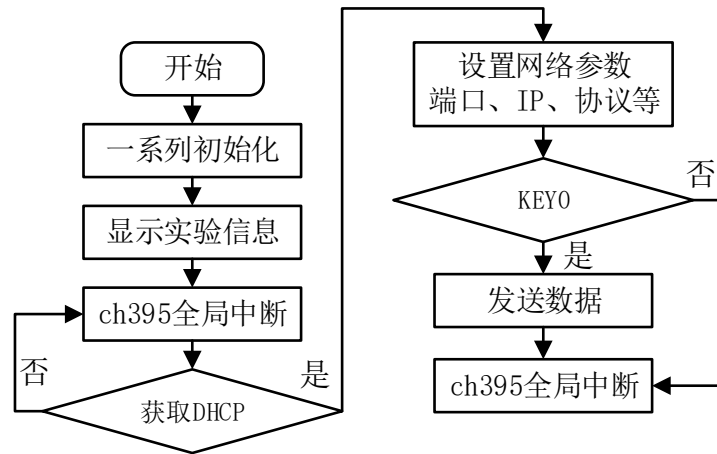


图 4.2.2.1.1 TCPServer 实验流程图

### 4.2.2.2 程序解析

关于 TCPServer 实验的程序解析，笔者着重讲解 ch395\_demo.c/h 文件，ch395\_demo.h 文件主要声明 ch395\_demo 函数提供外部文件使用，而 ch395\_demo.c 文件定义了三个函数，这些函数如下表所示：

函数	功能描述
ch395_show_mesg	显示实验信息
ch395_demo	例程测试

表 4.2.2.2.1 TCPServer 实验函数

下面是对本实验的 ch395\_demo.c 文件中的函数进行介绍，如下所示：

#### 1. 函数 ch395\_show\_mesg

该函数主要的作用是显示实验信息，如下源码所示：

```

/**
 * @brief      显示实验信息
 * @param      无
 * @retval     无
 */
void ch395_show_mesg(void)
{
    /* LCD 显示实验信息 */
    lcd_show_string(10, 10, 220, 32, 32, "STM32", RED);
    lcd_show_string(10, 47, 220, 24, 24, "CH395Q TCPServer", RED);
    lcd_show_string(10, 76, 220, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(10, 97, 200, 16, 16, "KEY0: Send", BLUE);
}
    
```

```

/* 串口输出实验信息 */
printf("\n");
printf("*****\r\n");
printf("STM32\r\n");
printf("CH395Q TCPServer\r\n");
printf("ATOM@ALIEN TEK\r\n");
printf("KEY0: Send\r\n");
printf("*****\r\n");
printf("\r\n");
}

```

此函数调用 lcd\_show\_string 和 printf 函数分别在 LCD 和串口上显示实验信息。

## 2. 函数 ch395\_demo

此函数主要设置 TCPServer 相关网络参数, 如配置协议、IP 及端口等信息, 如下源码所示:

```

/* 本地网络信息: IP 地址、网关地址、子网掩码和 MAC 地址 */
uint8_t ch395_ipaddr[4]    = {192,168,1,10};
uint8_t ch395_gw_ipaddr[4] = {192,168,1,1};
uint8_t ch395_ipmask[4]    = {255,255,255,0};
uint8_t ch395_macaddr[6]   = {0xB8,0xAE,0x1D,0x00,0x00,0x00};
/* 远程 IP 地址设置 */
uint8_t ch395_des_ipaddr[4] = {192,168,1,111};
static uint8_t socket0_send_buf[] = {"This is from CH395Q\r\n"};
static uint8_t socket0_recv_buf[1024];

ch395_socket cha95_sockct_sta[8];

/**
 * @brief      例程测试
 * @param      无
 * @retval     无
 */
void ch395_demo(void)
{
    uint8_t key = 0;
    ch395_show_mesg(); /* 显示信息 */
    do
    {
        ch395q_handler();
    }
    while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获取 DHCP*/

    /* 使能 socket 接口 */
    cha95_sockct_sta[0].socket_enable = CH395Q_ENABLE;
    /* 设置 socket 接口 */

```

```

cha95_sockct_sta[0].socket_index = CH395Q_SOCKET_0;
/* 设置目标 IP 地址 */
memcpy(cha95_sockct_sta[0].des_ip, ch395_des_ipaddr,
        sizeof(cha95_sockct_sta[0].des_ip));
/* 设置静态本地 IP 地址 */
memcpy(cha95_sockct_sta[0].net_config.ipaddr, ch395_ipaddr,
        sizeof(cha95_sockct_sta[0].net_config.ipaddr));
/* 设置静态网关 IP 地址 */
memcpy(cha95_sockct_sta[0].net_config.gwipaddr, ch395_gw_ipaddr,
        sizeof(cha95_sockct_sta[0].net_config.gwipaddr));
/* 设置静态子网掩码地址 */
memcpy(cha95_sockct_sta[0].net_config.maskaddr, ch395_ipmask,
        sizeof(cha95_sockct_sta[0].net_config.maskaddr));
/* 设置静态 MAC 地址 */
memcpy(cha95_sockct_sta[0].net_config.macaddr, ch395_macaddr,
        sizeof(cha95_sockct_sta[0].net_config.macaddr));
/* 源端口 */
cha95_sockct_sta[0].sour_port = 8080;
/* 设置协议 */
cha95_sockct_sta[0].proto = CH395Q_SOCKET_TCP_SERVER;
/* 发送数据 */
cha95_sockct_sta[0].send.buf = socket0_send_buf;
/* 发送数据大小 */
cha95_sockct_sta[0].send.size = sizeof(socket0_send_buf);
/* 接收数据缓冲区 */
cha95_sockct_sta[0].recv.buf = socket0_recv_buf;
/* 接收数据大小 */
cha95_sockct_sta[0].recv.size = sizeof(socket0_recv_buf);
/* 配置 socket 参数 */
ch395q_socket_config(&cha95_sockct_sta[0]);
while(1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)
    {
        ch395_send_data(0, (uint8_t *)socket0_send_buf,
                        strlen((char *)socket0_send_buf));
    }

    ch395q_handler();
}
}

```

首先笔者声明了 IP 地址、子网掩码、网关和 MAC 地址，这些网络参数是为了 DHCP 失败时可设置默认的网络信息，ch395\_des\_ipaddr 数组需要填写电脑的 IP 地址（必须填写正确），而 socket0\_send\_buf 和 socket0\_recv\_buf 分别为发送的数据和接收缓冲区，cha95\_socket\_sta 变量的类型为 ch395\_socket，该结构体描述每一个 socket 网络参数，该结构体如下所示：

```
typedef struct ch395q_socket_t
{
    uint8_t socket_enable;           /* Socket 使能 */
    uint8_t socket_index;           /* Socket 标号 */
    uint8_t proto;                   /* Socket 协议 */
    uint8_t des_ip[4];               /* 目的 IP 地址 */
    uint16_t des_port;               /* 目的端口 */
    uint16_t sour_port;              /* 源端口 */

    struct
    {
        uint8_t *buf;                /* 缓冲空间 */
        uint32_t size;                /* 缓冲空间大小 */
    } send;                           /* 发送缓冲 */

    struct
    {
        uint8_t *buf;                /* 缓冲空间 */
        uint32_t size;                /* 缓冲空间大小 */
    } recv;                           /* 接收缓冲 */

    struct
    {
        uint8_t ip[4];               /* IP 地址 */
        uint8_t gwip[4];             /* 网关 IP 地址 */
        uint8_t mask[4];             /* 子网掩码 */
        uint8_t dns1[4];             /* DNS 服务器 1 地址 */
        uint8_t dns2[4];             /* DNS 服务器 2 地址 */
    } net_info;                       /* 网络信息 */

    struct
    {
        uint8_t ipaddr[4];           /* IP 地址 32bit */
        uint8_t gwipaddr[4];         /* 网关地址 32bit */
        uint8_t maskaddr[4];         /* 子网掩码 32bit */
        uint8_t macaddr[6];          /* MAC 地址 48bit */
    } net_config;                     /* 网络配置信息 */
} ch395_socket;
```

此结构体笔者分为四个部分讲解，如下：

#### ① 描述每一个 socket 共同部分

```
uint8_t socket_enable;           /* Socket 使能 */
uint8_t socket_index;           /* Socket 标号 */
uint8_t proto;                  /* Socket 协议 */
uint8_t des_ip[4];              /* 目的 IP 地址 */
uint16_t des_port;              /* 目的端口 */
uint16_t sour_port;             /* 源端口 */
```

socket\_enable 为使能 socket，因为 CH395Q 可以开启 8 个 socket 接口，所以笔者使用这个成员变量描述 socket 接口是否打开；socket\_index 为 socket 的编号，例如：0~7 的 socket 接口；proto 为协议选择，描述这个 socket 接口使用那个协议，例如：UDP、TCPClient 和 TCPServer 协议类型；des\_ip[4] 设置目标 IP 地址，这个成员变量主要用作于 UDP 和 TCPClient 协议类型；des\_port 和 sour\_port 就是设置目标端口和源端口。

#### ② 收发结构体

```
struct
{
    uint8_t *buf;                /* 缓冲空间 */
    uint32_t size;               /* 缓冲空间大小 */
} send;                          /* 发送缓冲 */

struct
{
    uint8_t *buf;                /* 缓冲空间 */
    uint32_t size;               /* 缓冲空间大小 */
} recv;
```

每一个 socket 都可以收发数据，所以笔者在这里定义了收发结构体，用来描述发送缓冲区和发送缓冲区大小以及接收缓冲区和接收缓冲区大小。

#### ③ 网络信息（DHCP）

```
struct
{
    uint8_t ip[4];               /* IP 地址 */
    uint8_t gwip[4];            /* 网关 IP 地址 */
    uint8_t mask[4];            /* 子网掩码 */
    uint8_t dns1[4];            /* DNS 服务器 1 地址 */
    uint8_t dns2[4];            /* DNS 服务器 2 地址 */
} net_info;                     /* 网络信息 */
```

该结构体的成员变量是用来保存 DHCP 分配的网络信息，例如：IP 地址、子网掩码、网关以及 DNS 服务器地址。

#### ④ 网络配置信息（静态）

```
struct
{
    uint8_t ipaddr[4];           /* IP 地址 32bit */
    uint8_t gwipaddr[4];        /* 网关地址 32bit */
}
```

```
uint8_t maskaddr[4]; /* 子网掩码 32bit*/
uint8_t macaddr[6]; /* MAC 地址 48bit*/
} net_config; /* 网络配置信息 */
```

此结构体就是保存静态的网络信息，例如：DHCP 不成功时，我们设置 CH395Q 的网络信息为静态网络信息。

至此，ch395\_socket 结构体讲解完毕，总的来说：它就是用来描述每一个 socket 接口的网络参数等信息。接着回到 ch395\_demo 函数讲解，该函数调用 ch395\_show\_mesg 函数显示实验信息，并使用 do while 语句获取 DHCP 是否成功，如果获取成功，则 dhcp\_status 为 DHCP\_UP，否则为 DHCP\_DOWN，然后设置 socket 接口网络信息，例如：socket 编号、静态网络信息、端口号及协议类型等参数，最后等待连接，连接完成后按下 KEY0\_PRES 就可以发送数据了。

**注意：协议类型为 CH395Q\_SOCKET\_TCP\_SERVER。**

### 4.2.3 下载验证

下载代码完成后，打开网络调试助手，等待开发板的 LCD 出现下图所示的界面。



图 4.2.3.1 远端 IP 地址设置

接下来设置电脑端的网络调试助手，设置完成后点击网络调试助手的“连接”，操作完后的网络调试助手如下图所示：

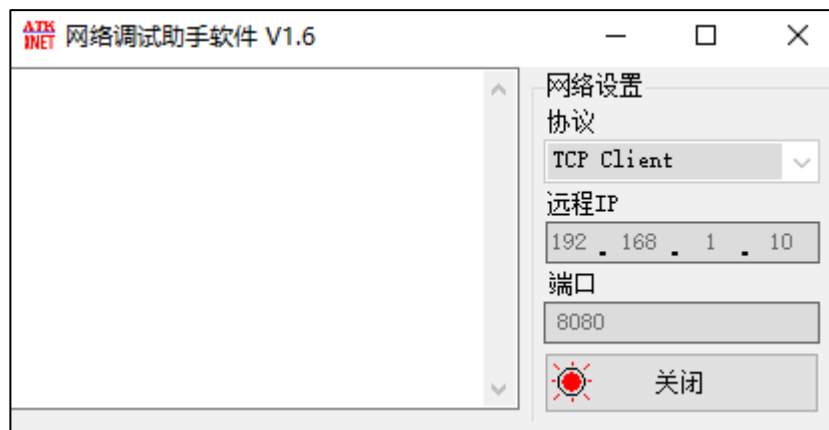


图 4.2.3.2 网络调试助手设置

设置完网络调试助手后在发送框填入要发送的数据，这里输入要发送的数据：ALIENTEK DATA，然后点击发送，这时串口调试助手显示接收的数据，接着通过按下 KEY0，向电脑端发送数据“This is from CH395Q\r\n”如下图所示，表明网络调试助手接收到开发板发送的数据，这里我们按了 13 次 KEY0，因此在网络调试助手上有 13 行数据。

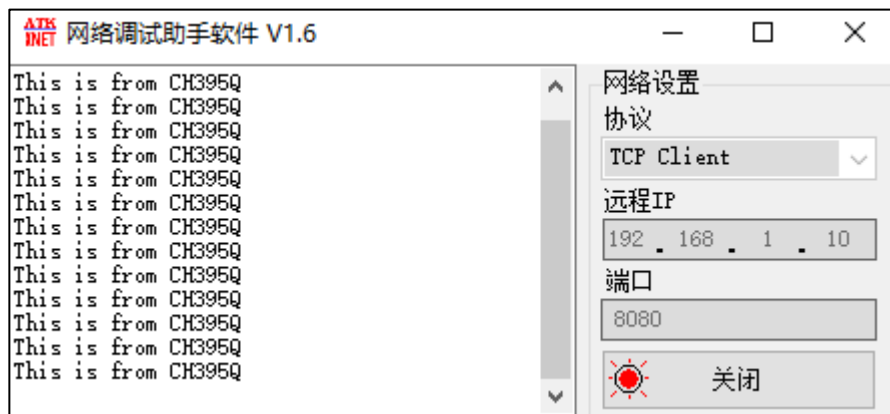


图 4.2.3.3 TCPServer 测试



## 第五章 WebServer 实验

本章采用 CH395Q 作为服务器在开发板上实现一个 WebServer。在本章中笔者通过移植 lwIP 下的 httpserver 实验来展示 WebServer，在浏览器输入开发板的 IP 地址来访问开发板，这时开发板会返回一个网页数据，浏览器根据这个网页数据构建一个网页。

本章我们分为几个部分讲解：

5.1 WebServer 简介

5.2 WebServer 实验

### 5.1 WebServer 简介

Web 服务器可以解析 HTTP 协议。当 Web 服务器接收到一个 HTTP 请求，会返回一个 HTTP 响应。为了处理一个请求，Web 服务器可以响应一个静态页面或图片，进行页面跳转，或者把动态响应的产生委托给一些其它的程序，例如 CGI 脚本，JSP 脚本，servlets，ASP 脚本，服务器端 JavaScript，或者一些其它的服务器端技术。无论它们的目的如何，这些服务器端的程序通常产生一个 HTML 的响应来让浏览器可以浏览。如下图所示。

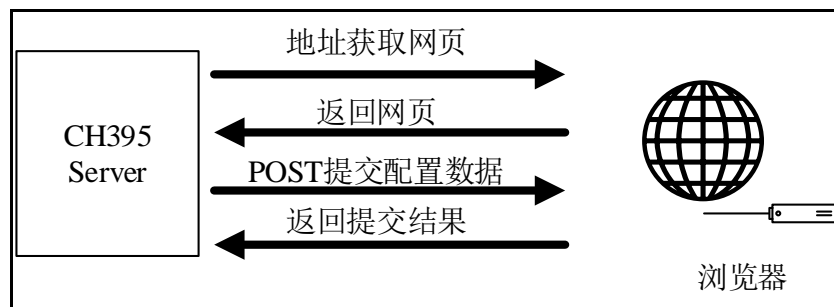


图 5.1.1 Webserver 原理

### 5.2 WebServer 实验

#### 5.2.1 硬件设计

##### 1. 例程功能

浏览器中输入开发板的 IP 地址来访问开发板，这时开发板会返回一个网页数据，浏览器根据这些网页数据构建一个网页界面。

该实验的实验工程，请参考《网络实验 5 CH395\_WEBSEVER 实验》。

#### 5.2.2 软件设计

##### 5.2.2.1 Webserver 函数解析

本实验是在《网络实验 4 CH395\_TCP 服务器实验》实验的基础上修改的，这里笔者重点讲解一下 ch395\_demo.c 文件下的函数，如下所示：

##### 1. 函数 ch395\_demo

TCPServer 服务器连接及初始化，该函数的原型如下源码所示：

```
void ch395_demo(void)
```

**函数形参：**

无。

**返回值：**

无。

## 2. 函数 ch395\_show\_mesg

显示实验信息，该函数的原型如下源码所示：

```
void ch395_show_mesg(void)
```

**函数形参：**

无。

**返回值：**

无。

## 3. 函数 ch395\_server\_netconn\_serve

接收一个 HTTP 连接，该函数的原型如下源码所示：

```
void ch395_server_netconn_serve(void)
```

**函数形参：**

无。

**返回值：**

无。

## 4. 函数 ch395\_data\_locate

寻找指定字符位置，该函数的原型如下源码所示：

```
char *ch395_data_locate(char *buf, char *name)
```

**函数形参：**

此函数是形参如下表所示：

函数形参	描述
buf	查询的数据
name	指定字符的名称

表 5.2.2.1.1 ch395\_data\_locate 函数形参描述

**返回值：**

返回网页字符的地址。

### 5.2.2.2 Webserver 服务器配置步骤

#### ① 配置 CH395 为 TCP 服务器模式

CH395Q 配置为 TCPServer 服务器的流程，请参考第四章的内容。

#### ② 接收 HTTP 协议请求报文

当网页输入 IP 地址时，CH395 会接收到一个 HTTP 协议的请求报文，接收完成之后系统调用函数 ch395\_server\_netconn\_serve 发送网页数据到浏览器中。

### 5.2.2.3 程序流程图

本实验的程序流程图，如下图所示：

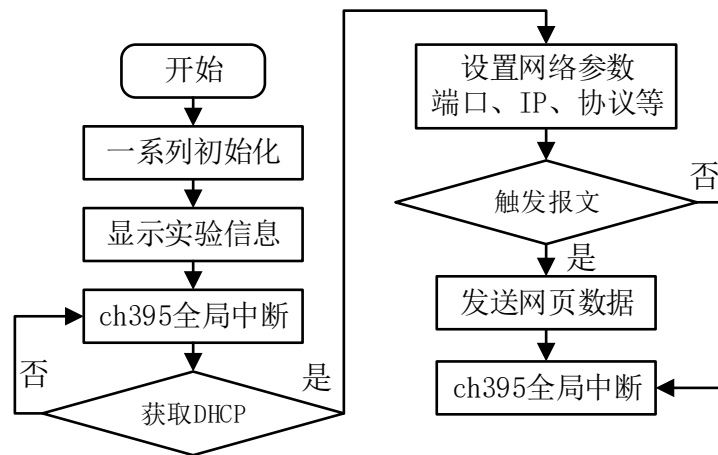


图 5.2.2.3.1 WebServer 程序流程图

#### 5.2.2.4 程序解析

打开 ch395\_demo.h 文件,在这个文件下我们声明了 ch395\_demo 函数提供给外部文件使用, ch395\_demo.c 文件如下所示:

##### ① 网络信息

```

/* 本地网络信息: IP 地址、网关地址、子网掩码和 MAC 地址 */
uint8_t ch395_ipaddr[4]      = {192,168,1,10};
uint8_t ch395_gw_ipaddr[4]  = {192,168,1,1};
uint8_t ch395_ipmask[4]     = {255,255,255,0};
uint8_t ch395_macaddr[6]    = {0xB8,0xAE,0x1D,0x00,0x00,0x00};
/* 远程 IP 地址设置 */
uint8_t ch395_des_ipaddr[4]  = {192,168,1,111};

```

##### ② 声明 HTTPS 头部

```

static const char http_html_hdr[] = "HTTP/1.1 200 OK\r\nContent-type:
                                     text/html\r\n\r\n";

```

##### ③ 网页数据

```

static const char http_index_html[] =
    "<!DOCTYPE html>\r\n"
    "<html xmlns='http://www.w3.org/1999/xhtml'>\r\n"
    "<head>\r\n"
    "<title></title>\r\n"
    "</head>\r\n"
    "<body>\r\n"
    "<div id='main'>\r\n"
    "<h3>正点原子学习网</h3>\r\n"
    "<iframe src='http://www.openedv.com/docs/index.html'
    width='1024px' height='1024px'></iframe>\r\n"
    "</div>\r\n"
    "</body>\r\n"
    "</html>\r\n";

```

由于笔者不怎么会编写 HTML 代码，所以上述的网页数据比较简单，它只能显示正点原子资料中心网页。

#### ④ 接收一个 HTTP 连接

```
/**
 * @brief      服务 HTTP 线程中接受的一个 HTTP 连接
 * @param      conn    netconn 控制块
 * @retval     无
 */
void ch395_server_netconn_serve(void)
{
    char *ptemp;
    /* 从端口读取数据，如果那里还没有数据，则阻塞。
     * 我们假设请求(我们关心的部分) */
    if (cha95_sockct_sta[0].recv.size != 0)
    {
        /* 这是一个 HTTP GET 命令吗?只检查前 5 个字符，因为
         * GET 还有其他格式，我们保持简单) */
        if (socket0_recv_buf[0] == 'G' &&
            socket0_recv_buf[1] == 'E' &&
            socket0_recv_buf[2] == 'T' &&
            socket0_recv_buf[3] == ' ' &&
            socket0_recv_buf[4] == '/')
        {
            start_html:
                /* 发送 HTML 标题
                 * 从大小中减去 1，因为我们没有在字符串中发送\0
                 * NETCONN_NOCOPY:我们的数据是常量静态的，所以不需要复制它 */
                ch395_send_data(0, (uint8_t *)http_html_hdr,
                               sizeof(http_html_hdr) - 1);

                /* 发送我们的 HTML 页面 */
                ch395_send_data(0, (uint8_t *)http_index_html,
                               sizeof(http_index_html) - 1);
        }
        else if(socket0_recv_buf[0] == 'P' && socket0_recv_buf[1] == 'O'
                && socket0_recv_buf[2] == 'S' && socket0_recv_buf[3] == 'T')
        {
            ptemp = ch395_data_locate((char *)socket0_recv_buf, "led1=");

            if (ptemp != NULL)
            {
                /* 查看 led1 的值。为 1 则灯亮，为 2 则灭，此值与 HTML 网页中设置有关 */
            }
        }
    }
}
```

```

        if (*ptemp == '1')
        {
            /* 点亮 LED1 */
        }
        else
        {
            /* 熄灭 LED1 */
        }

    }

    /* 查看 beep 的值。为 3 则灯亮，为 4 则灭，此值与 HTML 网页中设置有关 */
    ptemp = ch395_data_locate((char *)socket0_recv_buf, "beep=");

    if (ptemp != NULL )
    {
        if (*ptemp == '3')
        {
            /* 打开蜂鸣器 */
        }
        else
        {
            /* 关闭蜂鸣器 */
        }
    }
    goto start_html;
}

memset(socket0_recv_buf,0, sizeof(socket0_recv_buf));
delay_ms(100);
}

```

此函数可以分为两个部分讲解，第一部分是开发板接收到网页的“GET”请求，系统先发送 HTTPS 协议头部，接着发送网页数据，这样网页就可以显示了；第二部分是开发板接收到网页的“POST”请求，系统会根据 ch395\_data\_locate 函数判断请求的位置，并做出相应的动作。

## ⑥ 查询字符位置

```

/**
 * @brief      寻找指定字符位置
 * @param      buf    缓冲区指针
 * @param      name  寻找字符
 * @retval     返回字符的地址
 */
char *ch395_data_locate(char *buf, char *name)
{

```

```
char *p;
p = strstr((char *)buf, name);

if (p == NULL)
{
    return NULL;
}

p += strlen(name);
return p;
}
```

此函数非常简单，它根据接收的“POST”数据查询指定字符的位置。

### ⑦ 测试例程

```
/**
 * @brief      例程测试
 * @param      无
 * @retval     无
 */
void ch395_demo(void)
{
    ch395_show_mesg(); /* 显示信息 */

    do
    {
        ch395q_handler();
    }
    while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获 DHCP*/
    /* 使能 socket 接口 */
    cha95_sockct_sta[0].socket_enable = CH395Q_ENABLE;
    /* 设置 socket 接口 */
    cha95_sockct_sta[0].socket_index = CH395Q_SOCKET_0;
    /* 设置目标 IP 地址 */
    memcpy(cha95_sockct_sta[0].des_ip, ch395_des_ipaddr,
           sizeof(cha95_sockct_sta[0].des_ip));
    /* 设置静态本地 IP 地址 */
    memcpy(cha95_sockct_sta[0].net_config.ipaddr, ch395_ipaddr,
           sizeof(cha95_sockct_sta[0].net_config.ipaddr));
    /* 设置静态网关 IP 地址 */
    memcpy(cha95_sockct_sta[0].net_config.gwipaddr, ch395_gw_ipaddr,
           sizeof(cha95_sockct_sta[0].net_config.gwipaddr));
    /* 设置静态子网掩码地址 */
    memcpy(cha95_sockct_sta[0].net_config.maskaddr, ch395_ipmask,
           sizeof(cha95_sockct_sta[0].net_config.maskaddr));
}
```

```
/* 设置静态 MAC 地址 */
memcpy(ch395_sockct_sta[0].net_config.macaddr, ch395_macaddr,
        sizeof(ch395_sockct_sta[0].net_config.macaddr));

/* 源端口 */
ch395_sockct_sta[0].sour_port = 80;

/* 设置协议 */
ch395_sockct_sta[0].proto = CH395Q_SOCKET_TCP_SERVER;

/* 接收数据缓冲区 */
ch395_sockct_sta[0].recv.buf = socket0_recv_buf;

/* 接收数据大小 */
ch395_sockct_sta[0].recv.size = sizeof(socket0_recv_buf);

/* 配置 socket 参数 */
ch395q_socket_config(&ch395_sockct_sta[0]);
while(1)
{
    ch395_server_netconn_serve();
    ch395q_handler();
}
}
```

注意：源端口必须设置为 80 端口，因为 80 端口是为 HTTP 超文本传输协议开放的端口。

### 5.2.3 下载验证

打开网页输入 IP 地址，如下图所示。



图 5.2.3.1 WebServer 效果图

可以看出，浏览器比喻成一个 HTML 的编译器，而 CH395Q 传输的网页数据就是编译器可识别的代码段。

## 第六章 NTP 实时时间实验

NTP 网络时间协议是基于 UDP 协议上实现，它用于网络时间同步，使网络中的计算机时钟同步到 UTC，再配合各个时区的偏移调整就能实现精准同步对时功能。

本章我们分为如下几个部分讲解：

6.1 NTP 简介

6.2 NTP 实验

### 6.1 NTP 简介

NTP 服务器是用来使计算机时间同步化的一种协议，它可以使计算机对其服务器或时钟源（如石英钟，GPS 等等）做同步化，它还可以提供高精度的时间校正，且可介由加密确认的方式来防止恶毒的协议攻击。NTP 协议的数据报文格式如下图所示：

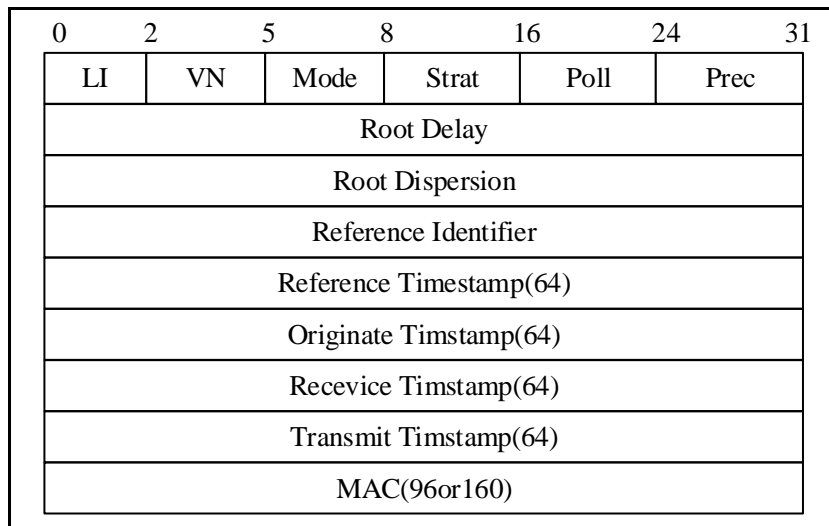


图 6.1.1 NTP 报文格式

从上图可知，NTP 报文是由多个字段组成，每一个字段所获取的功能都不一样，一般获取实时时间只使用 VN 字段和 Mode 字段即可完成，其他的字段使用请读者查阅相关的 NTP 协议资料。

NTP 数据报文格式的各个字段的作用，如下表所示：

字节段	描述
LI: 2 比特	11 为告警状态，表示时钟未被同步
VN: 3 比特	表示 NTP 的版本号
Mode: 3 比特	0: 未定义、1: 等体模式、2: 被动对等体模式、3: 客户模式、4: 服务器模式、5: 广播模式
Strat: 8 比特	表示系统时钟的层数，取值范围为 1~16
Poll: 8 比特	表示轮询时间，即两个连续 NTP 报文之间的时间间隔
Prec: 8 比特	表示系统时钟的精度
Root Delay: 32 比特	表示本地到主参考时钟源的往返时间
Root Dispersion: 32 比特	表示系统时钟相对于主参考时钟的最大误差



Reference Identifier: 32 比特	表示参考时钟源的标识
Reference Timestamp: 64 比特	表示系统时钟最后一次被设定或更新的时间
Originate Timestamp: 64 比特	表示 NTP 请求报文离开发送端时发送端的本地时间
Receive Timestamp: 64 比特	表示 NTP 请求报文到达接收端时接收端的本地时间
Transmit Timestamp: 64 比特	表示应答报文离开应答者时应答者的本地时间
Authenticator: 96 比特	表示验证信息我们怎么获取阿里云 NTP 实时时间数据呢

表 6.1.2 NTP 报文字段描述

从上表可知，NTP 报文的字段非常多，这些字段并不是每一个都必须设置的，请大家根据项目的需要来构建 NTP 请求报文。下面笔者使用网络调试助手制作一个简单的 NTP 实验，如下图所示：

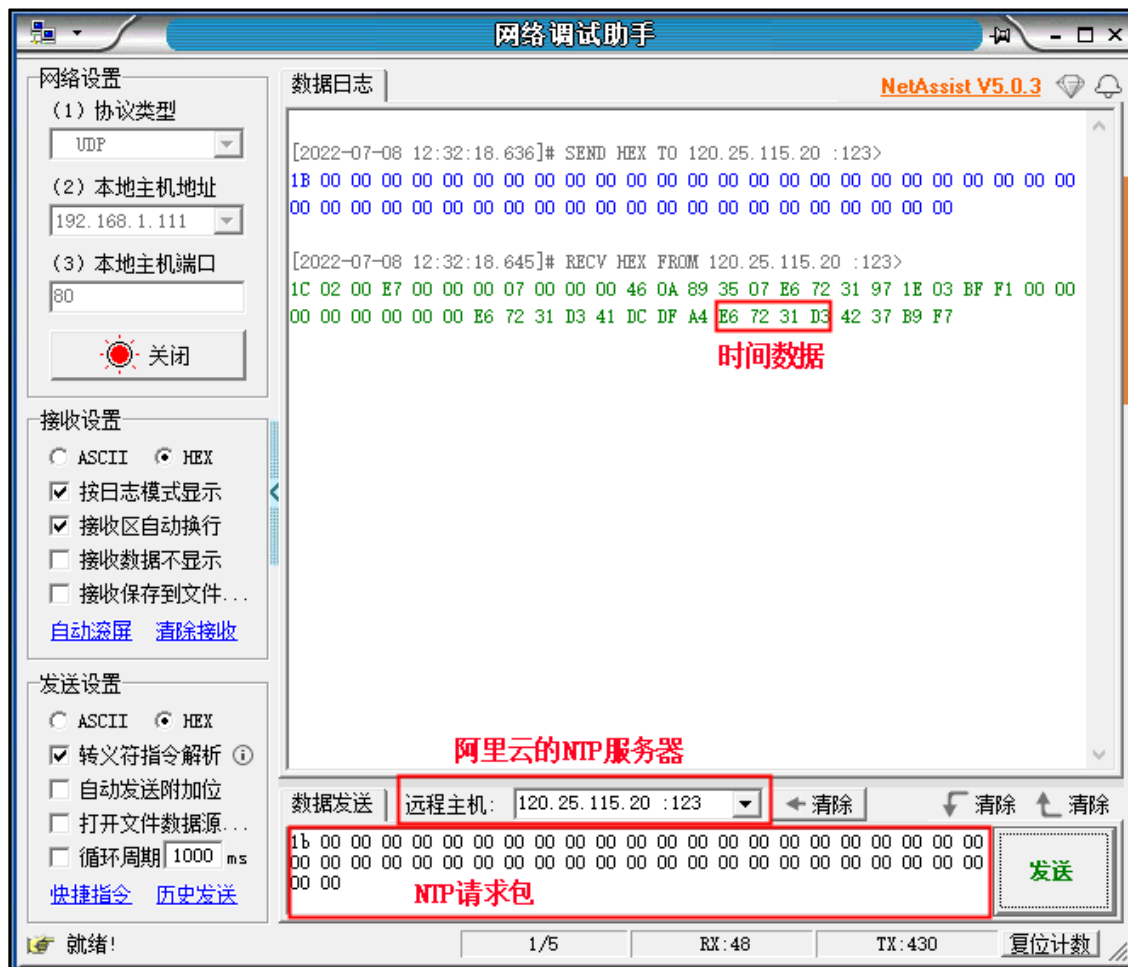


图 6.1.1 获取阿里云 NTP 数据

这里笔者使用 UDP 协议连接阿里云 NTP 服务器，该服务器的域名为 `ntp1.aliyun.com`，我们可在在线域名网页解析该域名，并使用网络调试助手连接阿里云 NTP 服务器。这里笔者只设置 NTP 报文的 VN 字段和 Mode 字段，该些字段组成后可以填入 `0xa3`（版本 4）、`0x1b`（版本 3）、`0x13`（版本 2）和 `0x0b`（版本 1），其他字段可设置为 0。上图中笔者使用的是版本 3 来获取阿里云 NTP 时间信息，发送 NTP 报文完成之后阿里云 NTP 服务器会返回一些数据，这些数据包含了当前时间数据，例如上图中 `0xE67231D3` 十六进制就是 NTP 服务器返回的时间戳，这个数据是十六进制展示的，所以我们把这个十六进制转成是十进制（3866243539），接着减去 1900-1970 的时间差（2208988800 秒）就等于当前的秒数（1657254739），最后把这个秒数

进行时间戳转换，转换完成后可得到实时时间信息。在浏览器上打开 <https://tool.lu/timestamp/> 网站，从这个网页上计算秒数，并把它转换成实时时间，如下图所示：

现在：	1657255021	控制：	■ 停止
时间戳	1657254739	秒(s) ▾	转换 >> 2022-07-08 12:32:19 北京时间
时间	2022-07-08 12:36:31	北京时间	转换 >> <input type="text"/> 秒(s) ▾

图 6.1.2 秒转成时间

获取 NTP 实时时间需要哪些步骤了，如下所示：

- ① 连接阿里云 NTP 服务器，我们以 UDP 协议为例。
- ② 使用开发板发送 NTP 报文到阿里云 NTP 服务器中。
- ③ 获取阿里云 NTP 服务器的数据，取第 40 位到 43 位的十六进制。
- ④ 把 40 位到 43 位的十六进制转成十进制。
- ⑤ 把十进制数减去 1900-1970 的时间差（2208988800 秒）。
- ⑥ 数值转成年月日时分秒。

根据上述的流程，我们可在工程中构建 NTP 报文并且传输到阿里云服务器，传输完成之后接收阿里云服务器应答的数据，最后对这些数据从③~⑥进行转换。

## 6.2 NTP 实验

### 6.2.1 硬件设计

#### 1. 例程功能

使用 UDP 协议连接阿里云的 NTP 服务器，并周期发送 NTP 请求报文，发送完成之后对阿里云 NTP 服务器返回的数据进行解析，把它转换成实时时间信息。

该实验的实验工程，请参考《网络实验 6 CH395\_NTP 网络时间实验》。

### 6.2.2 软件设计

#### 6.2.2.1 NTP 函数解析

本实验是在《网络实验 2 CH395\_UDP 实验》实验的基础上修改的，这里笔者重点讲解一下 ch395\_deno.c 文件下的 ch395\_ntp\_client\_init、ch395\_get\_seconds\_from\_ntp\_server 和 ch395\_calc\_date\_time 函数，如下所示：

##### 1. 函数 ch395\_ntp\_client\_init

构建 NTP 请求报文，该函数的原型如下源码所示：

```
void ch395_ntp_client_init(void)
```

函数形参：

无。

返回值：

无。

##### 2. 函数 ch395\_get\_seconds\_from\_ntp\_server

从 NTP 服务器获取时间，该函数的原型如下源码所示：

```
void ch395_get_seconds_from_ntp_server(uint8_t *buf, uint16_t idx)
```

**函数形参：**

此函数只有 2 个形参如下表所示：

参数	描述
buf	数据缓存
idx	保存数据起始位置

表 6.2.2.1.1 函数 ch395\_get\_seconds\_from\_ntp\_server()形参描述

**函数返回值：**

无。

**3. 函数 ch395\_calc\_date\_time**

计算日期时间，并转换成 UTC 世界标准时间，该函数的原型如下源码所示：

```
void ch395_calc_date_time(unsigned long long time)
```

**函数形参：**

此函数的形参如下表所示：

参数	描述
time	秒数

表 6.2.2.1.2 函数 ch395\_calc\_date\_time()形参描述

**函数返回值：**

无。

**6.2.2.2 NTP 配置步骤**
**① 配置 CH395 为 UDP 模式**

CH395Q 配置为 UDP 方式，请参考第二章的内容。

**② 制作 NTP 请求报文**

调用函数 ch395\_ntp\_client\_init 制作 NTP 请求报文，周期发送 NTP 请求报文。

**③ 处理 NTP 返回的信息**

调用函数 ch395\_get\_seconds\_from\_ntp\_server 处理 NTP 服务器返回的数据，这里笔者取 40 位到 43 位的数据，并且转换成十进制。

**④ 计算日期时间**

总秒数需要减去 1900-1970 的时间差（2208988800 秒），减去之后的秒数就是当前时间的总秒数，利用算法把秒数转换成当前时间。

**6.2.2.3 程序流程图**

本实验的程序流程图，如下图所示：

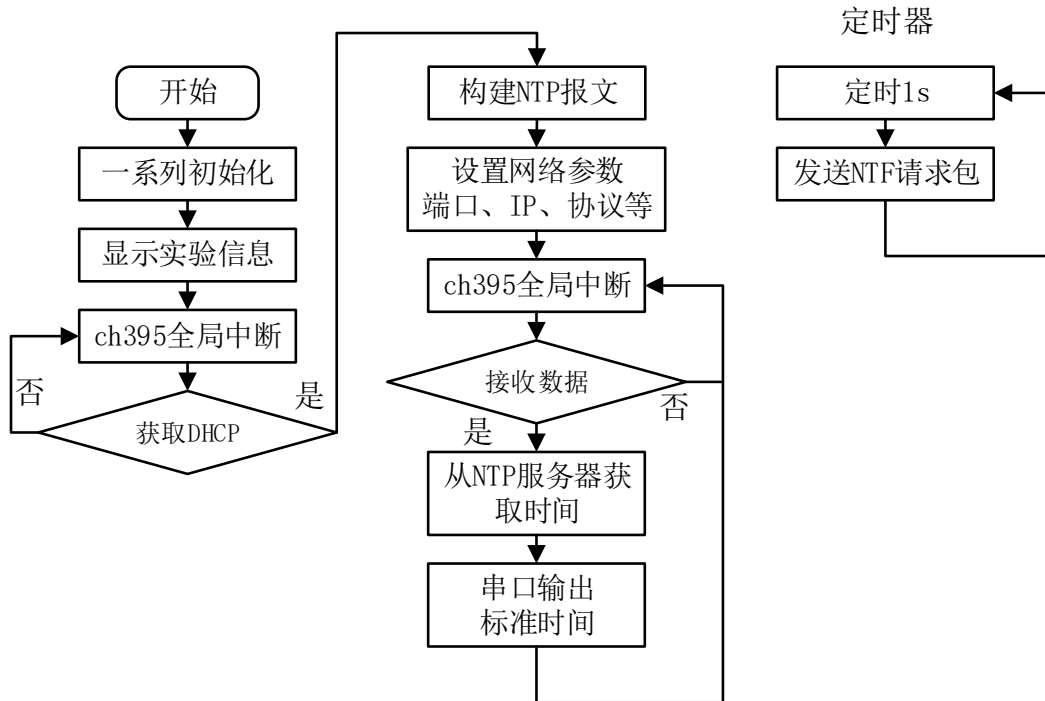


图 6.2.2.3.1 NTP 实验流程图

#### 6.2.2.4 程序解析

本次实验重点的内容是在 ch395\_demo.c/h 文件中，下面笔者分别地讲解这两个文件实现的内容，如下所示：

##### 1. 文件 ch395\_demo.h

打开 ch395\_demo.h 文件，在这个文件中笔者定义了 NPTformat 和 DateTime 结构体，NPTformat 结构体的成员变量与 NTP 的报文结构的字段是一一对应的，而 DateTime 结构体主要保存转换后标准时间的年月日时分秒等信息，它还声明了 ch395\_demo.c 文件中的函数，提供给外部文件使用，如下源码所示：

```

#ifndef __CH395_DEMO_H
#define __CH395_DEMO_H
#include "../SYSTEM/sys/sys.h"

typedef struct _NPTformat
{
    char    version;           /* 版本号 */
    char    leap;              /* 时钟同步 */
    char    mode;              /* 模式 */
    char    stratum;           /* 系统时钟的层数 */
    char    poll;              /* 更新间隔 */
    signed char    precision;   /* 精密度 */
    unsigned int    rootdelay;   /* 本地到主参考时钟源的往返时间 */
    unsigned int    rootdisp;    /* 统时钟相对于主参考时钟的最大误差 */
}
    
```

```

char    refid;                /* 参考识别码 */
unsigned long long  reftime;   /* 参考时间 */
unsigned long long  org;       /* 开始的时间戳 */
unsigned long long  rec;       /* 收到的时间戳 */
unsigned long long  xmt;       /* 传输时间戳 */
} NTPformat;

typedef struct _DateTime       /*此结构体定义了 NTP 时间同步的相关变量*/
{
    int  year;                 /* 年 */
    int  month;                /* 月 */
    int  day;                  /* 天 */
    int  hour;                 /* 时 */
    int  minute;               /* 分 */
    int  second;               /* 秒 */
} DateTime;

#define SECS_PERDAY    86400UL    /* 一天中的几秒钟 = 60*60*24 */
#define UTC_ADJ_HRS    8          /* SEOUL : GMT+8 (东八区北京) */
#define EPOCH          1900       /* NTP 起始年 */

void ch395_demo(void);          /* 例程测试 */
void ch395_get_seconds_from_ntp_server(uint8_t *buf, uint16_t idx);
#endif
    
```

从上述源码可以看出，ch395\_demo.h 文件主要声明了一些结构体和变量，这些结构体和宏定义会在 ch395\_demo.c 文件中调用。

## 2. 文件 ch395\_demo.c

### ① 函数 ch395\_ntp\_client\_init

构建 NTP 请求报文，如下源码所示：

```

/**
 * @brief      初始化 NTP Client 信息
 * @param      无
 * @retval     无
 */
void ch395_ntp_client_init(void)
{
    uint8_t flag;

    g_ntpformat.leap = 0;        /* leap indicator */
    g_ntpformat.version = 3;    /* version number */
    g_ntpformat.mode = 3;       /* mode */
    g_ntpformat.stratum = 0;    /* stratum */
    g_ntpformat.poll = 0;       /* poll interval */
}
    
```

```

g_ntpformat.precision = 0;          /* precision */
g_ntpformat.rootdelay = 0;          /* root delay */
g_ntpformat.rootdisp = 0;          /* root dispersion */
g_ntpformat.refid = 0;              /* reference ID */
g_ntpformat.reftime = 0;            /* reference time */
g_ntpformat.org = 0;                /* origin timestamp */
g_ntpformat.rec = 0;                /* receive timestamp */
g_ntpformat.xmt = 0;                /* transmit timestamp */

flag = (g_ntpformat.version << 3) + g_ntpformat.mode; /* one byte Flag */
memcpy(socket0_ntp_message, (void const *)(&flag), 1);
btim_timx_int_init(9999, 7199);
}

```

从上述源码可以看出，笔者只设置 version、mode 字段，其他字段都设置为 0，接着对这两个字段进行移位（ $0011 \ll 3(\text{version}) + 0011(\text{mode}) = 0x1b$ ）可得到 0x1b 十六进制数值，并把它保存在 socket0\_ntp\_message 数组当中，开启定时器周期发送 NTP 请求报文。该定时器服务函数如下所示：

```

/**
 * @brief      回调函数，定时器中断服务函数调用
 * @param      无
 * @retval     无
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim == (&timx_handler))
    {
        ch395_udp_send_data(ch395_sockct_sta[0].send.buf,
                            ch395_sockct_sta[0].send.size,
                            ch395_sockct_sta[0].des_ip,
                            ch395_sockct_sta[0].des_port,
                            ch395_sockct_sta[0].socket_index); /* 发送一个 NTP 包 */
    }
}

```

此定时服务函数的工作是 1s 发送一次 NTP 请求包。

## ② 函数 ch395\_get\_seconds\_from\_ntp\_server

处理 NTP 服务器返回的数据，如下源码所示：

```

/**
 * @brief      从 NTP 服务器获取时间
 * @param      buf: 存放缓存
 * @param      idx: 定义存放数据起始位置
 * @retval     无
 */
void ch395_get_seconds_from_ntp_server(uint8_t *buf, uint16_t idx)

```

```
{
    unsigned long long atk_seconds = 0;
    uint8_t i = 0;

    for (i = 0; i < 4; i++) /* 获取 40~43 位的数据 */
    {
        /* 把 40~43 位转成 16 进制再转成十进制 */
        atk_seconds = (atk_seconds << 8) | buf[idx + i];
    }

    /* 减去减去 1900-1970 的时间差 (2208988800 秒) */
    atk_seconds -= NTP_TIMESTAMP_DELTA;

    ch395_calc_date_time(atk_seconds); /* 由 UTC 时间计算日期 */
}
```

该函数主要完成三个操作：

- ① 从 NTP 服务器返回的数据中获取 40~43 位的数据，并把它转换成十进制数值。
- ② 减去减去 1900-1970 的时间差 (2208988800 秒)。
- ③ 调用 ch395\_calc\_date\_time 函数计算日期。

### ③ 函数 ch395\_calc\_date\_time

利用总秒数换算成日期数据，如下源码所示：

```
/**
 * @brief    计算日期时间
 * @param    time UTC 世界标准时间
 * @retval    无
 */
void ch395_calc_date_time(unsigned long long time)
{
    unsigned int Pass4year;
    int hours_per_year;
    if (time <= 0)
    {
        time = 0;
    }

    g_nowdate.second = (int)(time % 60); /* 取秒时间 */
    time /= 60;

    g_nowdate.minute = (int)(time % 60); /* 取分钟时间 */
    time /= 60;

    g_nowdate.hour = (int)(time % 24); /* 小时数 */
    /* 取过去多少个四年，每四年有 1461*24 小时 */
    Pass4year = ((unsigned int)time / (1461L * 24L));
```

```
g_nowdate.year = (Pass4year << 2) + 1970;          /* 计算年份 */

time %= 1461 * 24;                                  /* 四年中剩下的小时数 */

for (;;)      /* 校正闰年影响的年份，计算一年中剩下的小时数 */
{
    hours_per_year = 365 * 24;                      /* 一年的小时数 */

    if ((g_nowdate.year & 3) == 0)                  /* 判断闰年 */
    {
        hours_per_year += 24;                      /* 是闰年，一年则多 24 小时，即一天 */
    }

    if (time < hours_per_year)
    {
        break;
    }

    g_nowdate.year++;
    time -= hours_per_year;
}

time /= 24;     /* 一年中剩下的天数 */

time++;        /* 假定为闰年 */

if ((g_nowdate.year & 3) == 0)    /* 校正闰年的误差，计算月份，日期 */
{
    if (time > 60)
    {
        time--;
    }
    else
    {
        if (time == 60)
        {
            g_nowdate.month = 1;
            g_nowdate.day = 29;
            return ;
        }
    }
}
```



```

}

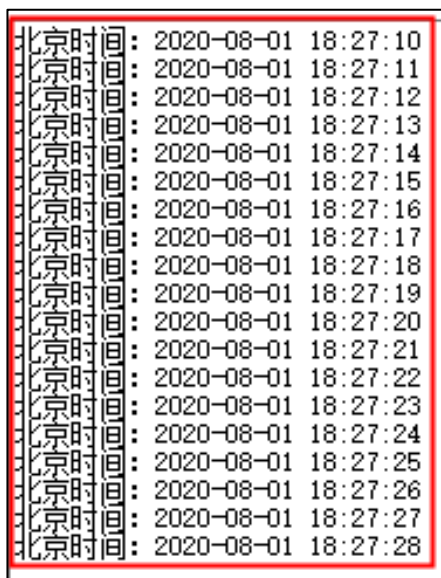
/* 计算月日 */
for (g_nowdate.month = 0; Days[g_nowdate.month] < time; g_nowdate.month++)
{
    time -= Days[g_nowdate.month];
}
g_nowdate.day = (int)(time);
return;
}

```

计算闰年等信息，最后把年月日时分秒等数据存储在 g\_nowdate 结构体当中。

### 6.2.3 下载验证

编译下载到开发板，打开串口调试助手如下图所示：



```

北京时间: 2020-08-01 18:27:10
北京时间: 2020-08-01 18:27:11
北京时间: 2020-08-01 18:27:12
北京时间: 2020-08-01 18:27:13
北京时间: 2020-08-01 18:27:14
北京时间: 2020-08-01 18:27:15
北京时间: 2020-08-01 18:27:16
北京时间: 2020-08-01 18:27:17
北京时间: 2020-08-01 18:27:18
北京时间: 2020-08-01 18:27:19
北京时间: 2020-08-01 18:27:20
北京时间: 2020-08-01 18:27:21
北京时间: 2020-08-01 18:27:22
北京时间: 2020-08-01 18:27:23
北京时间: 2020-08-01 18:27:24
北京时间: 2020-08-01 18:27:25
北京时间: 2020-08-01 18:27:26
北京时间: 2020-08-01 18:27:27
北京时间: 2020-08-01 18:27:28

```

图 6.2.3.1 串口打印标准时间

## 第七章 基于 MQTT 协议连接 OneNET 服务器

本章主要介绍 CH395Q 如何通过 MQTT 协议将设备连接 OneNET 平台, 并实现端云控制。本章我们分为几个部分讲解。

### 7.1 MQTT 协议简介

#### 7.2 配置 OneNET 平台

#### 7.3 工程配置

#### 7.4 基于 OneNET 平台 MQTT 实验

### 7.1 MQTT 协议简介

#### (1) MQTT 是什么?

MQTT (Message Queuing Telemetry Transport, 消息队列遥测传输协议), 是一种基于发布/订阅 (Publish/Subscribe) 模式的轻量级通讯协议, 该协议构建于 TCP/IP 协议上, 由 IBM 在 1999 年发布, 目前最新版本为 v3.1.1。MQTT 最大的优点在于可以以极少的代码和有限的带宽, 为远程设备提供实时可靠的消息服务。做为一种低开销、低带宽占用的即时通讯协议, MQTT 在物联网、小型设备、移动应用等方面有广泛的应用, MQTT 协议属于应用层。

#### (2) MQTT 协议特点

MQTT 是一个基于客户端与服务器的消息发布/订阅传输协议。MQTT 协议是轻量、简单开放和易于实现的, 这些特点使它适用范围非常广泛。在很多情况下, 包括受限境中, 如: 机器与机器 (M2M) 通信和物联网 (IoT)。其在, 通过卫星链路通信传感器、医疗设备、智能家居、及一些小型化设备中已广泛使用。

#### (3) MQTT 协议原理及实现方式

客户端和服务器端 MQTT 协议中有三种身份: 发布者 (Publish)、代理 (Broker) (服务器)、订阅者 (Subscribe)。其中, 消息的发布者和订阅者都是客户端, 消息代理是服务器, 消息发布者可以同时是订阅者, 如下图所示:

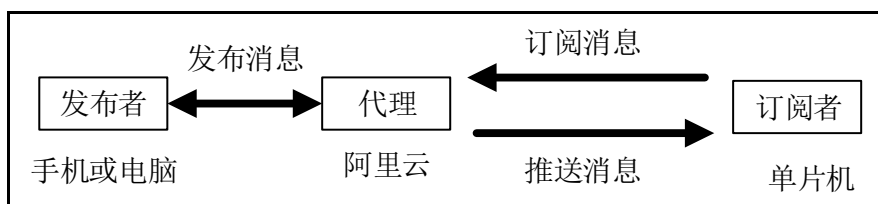


图 7.1.1 MQTT 订阅和发布过程

MQTT 传输的消息分为: 主题 (Topic) 和消息的内容 (payload) 两部分。

**Topic:** 可以理解为消息的类型, 订阅者订阅 (Subscribe) 后, 就会收到该主题的消息内容 (payload)。

**Payload:** 可以理解为消息的内容, 是指订阅者具体要使用的内容。

#### 7.1.1 MQTT 协议实现原理

1. 要在客户端与代理服务端建立一个 TCP 连接, 建立连接的过程是由客户端主动发起的, 代理服务一直是处于指定端口的监听状态, 当监听到有客户端要接入的时候, 就会立刻去处理。客户端在发起连接请求时, 携带客户端 ID、账号、密码 (无账号密码使用除外, 正式项目不会允许这样)、心跳间隔时间等数据。代理服务收到后检查自己的连接权限配置中是否允许该账

号密码连接，如果允许则建立会话标识并保存，绑定客户端 ID 与会话，并记录心跳间隔时间（判断是否掉线和启动遗嘱时用）和遗嘱消息等，接着回发连接成功确认消息给客户端，客户端收到连接成功的确认消息后，进入下一步（通常是开始订阅主题，如果不需要订阅则跳过）。如下图所示：

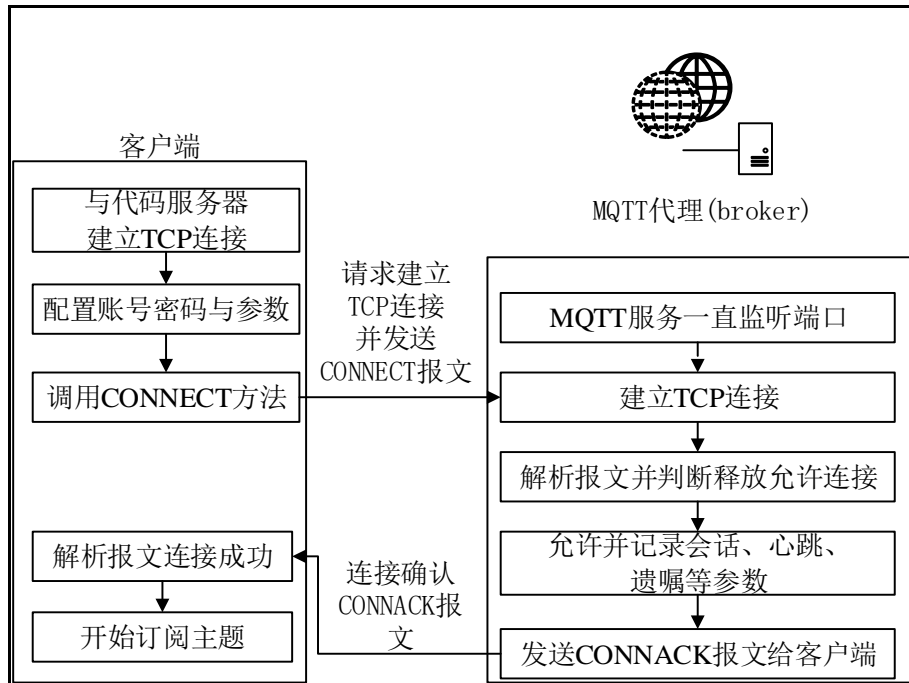


图 7.1.1.1 客户端与代理服务器建立连接示意图

2. 客户端将需要订阅的主题经过 SUBSCRIBE 报文发送给代理服务，代理服务则将这个主题记录到该客户端 ID 下（以后有这个主题发布就会发送给该客户端），然后回复确认消息 SUBACK 报文，客户端接到 SUBACK 报文后知道已经订阅成功，则处于等待监听代理服务推送的消息，也可以继续订阅其他主题或发布主题，如下图所示：

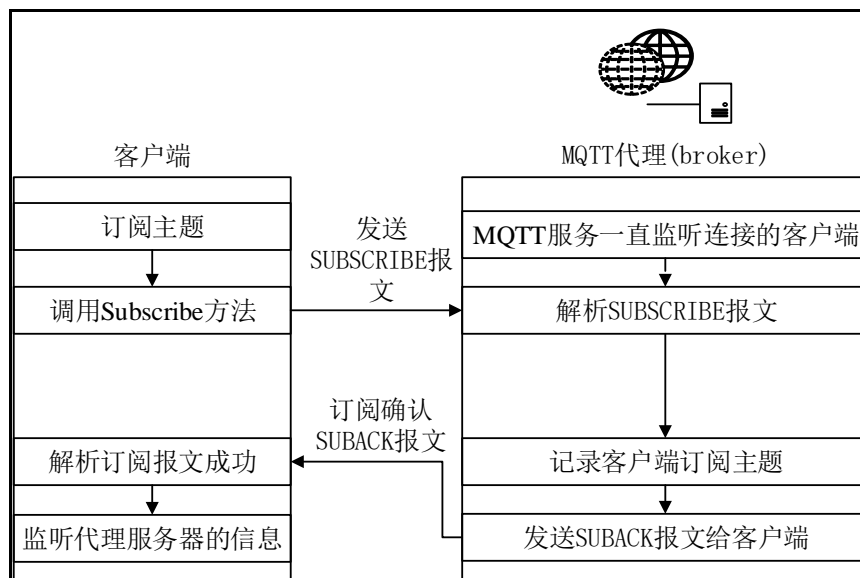


图 7.1.1.2 客户端向服务器订阅示意图

3. 当某一客户端发布一个主题到代理服务后，代理服务先回复该客户端收到主题的确切消息，该客户端收到确认后就可以继续自己的逻辑了。但这时主题消息还没有发给订阅了这个主

题的客户端，代理要根据质量级别（QoS）来决定怎样处理这个主题。所以这里充分体现了是 MQTT 协议是异步通信模式，不是立即端到端反应的，如下图所示：

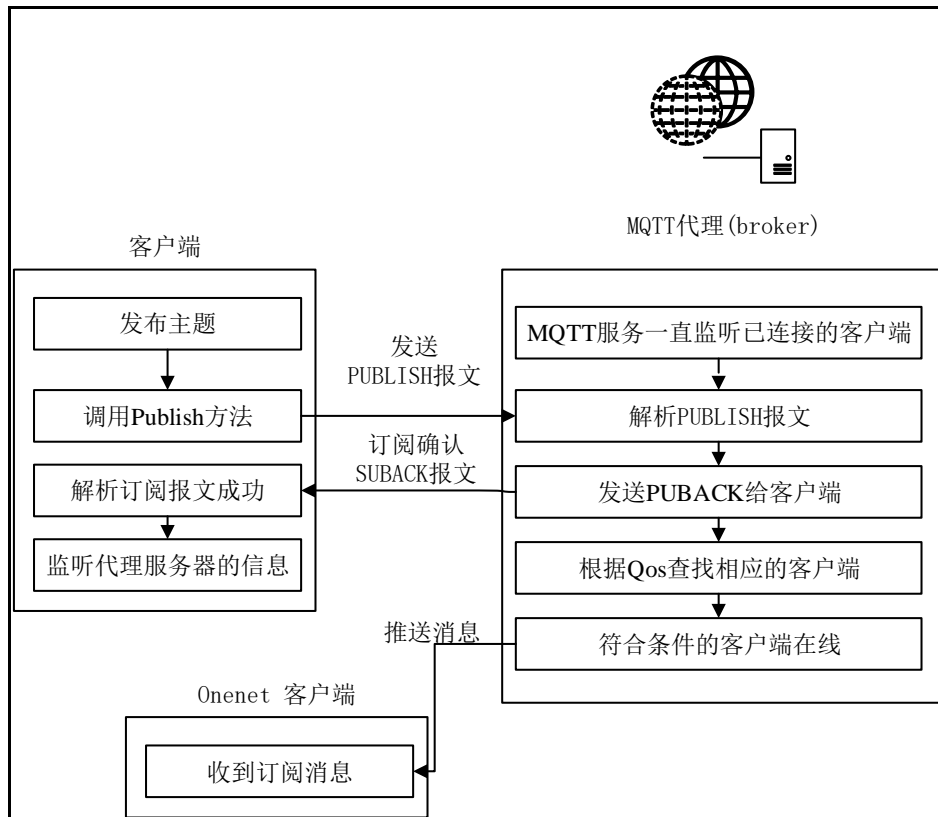


图 7.1.1.3 客户端向代理服务器发送主题

如果发布和订阅时的质量级别 QoS 都是至多一次，那代理服务则检查当前订阅这个主题的客户端是否在线，在线则转发一次，收到与否不再做任何处理。这种质量对系统压力最小。

如果发布和订阅时的质量级别 QoS 都是至少一次，那要保证代理服务和订阅的客户端都有成功收到才可以，否则会尝试补充发送（具体机制后面讨论）。这也可能会出现同一主题多次重复发送的情况。这种质量对系统压力较大。

如果发布和订阅时的质量级别 QoS 都是只有一次，那要保证代理服务和订阅的客户端都有成功收到，并只收到一次不会重复发送。这种质量级别对系统压力最大。

## 7.1.2 移植 MQTT 协议

若 CH395Q 以太网芯片实现 MQTT 协议的应用，则必须在工程中添加 MQTT 代码包，该库可在 <http://mqtt.org/> 网址下载。这里笔者以《网络实验 3 CH395\_TCP 客户端实验》实验为基础，在该工程 Middlewares 目录下新建 MQTT 文件夹，用来保存 MQTT 代码包 MQTTPacket\src 的.c/h 文件，该文件夹结构如下所示：

MQTTConnect.h	C++ Header file
MQTTConnectClient.c	C 文件
MQTTConnectServer.c	C 文件
MQTTDeserializePublish.c	C 文件
MQTTFormat.c	C 文件
MQTTFormat.h	C++ Header file
MQTTPacket.c	C 文件
MQTTPacket.h	C++ Header file
MQTTPublish.h	C++ Header file
MQTTSerializePublish.c	C 文件
MQTTSubscribe.h	C++ Header file
MQTTSubscribeClient.c	C 文件
MQTTSubscribeServer.c	C 文件
MQTTUnsubscribe.h	C++ Header file
MQTTUnsubscribeClient.c	C 文件
MQTTUnsubscribeServer.c	C 文件
StackTrace.h	C++ Header file

图 7.1.2.1 新建 MQTT 文件

我们在工程新建 Middlewares/MQTT 分组，在此分组下添加 MQTT 文件，该分组如下图所示。

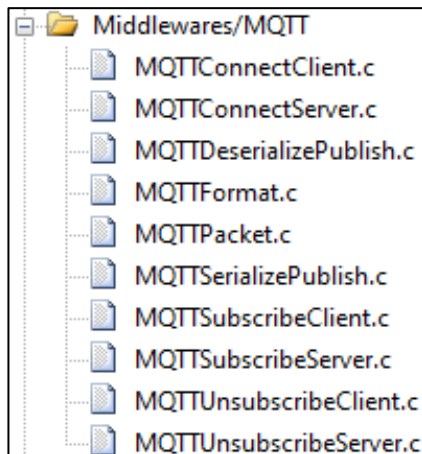


图 7.1.2.2 添加 MQTT 文件

## 7.2 配置 OneNET 平台

### 配置 OneNET 服务器步骤：

第一步：打开 OneNET 服务器并注册账号，注册之后在主界面下打开产品服务页面的 MQTT 物联网套件，如下图所示：



图 7.2.1 MQTT 物联网套件（新版）

第二步：点击上图的“立刻使用”选项，跳到其他页面之后点击“添加产品”选项，此时该页面会弹出产品信息小界面，这小界面如下图所示：

产品信息

\* 产品名称:

1-16个字符

\* 产品行业:

请选择

\* 产品类别:

请选择

请选择

请选择

产品简介:

1-200个字符

图 7.2.2 填写产品相关信息

上图中，笔者重点填写联网方式和设备接入协议的选项，它们分别选择移动蜂窝网络及MQTT 协议接入，至于其他选项请大家根据爱好选择。创建 MQTT 产品之后，用户可以得到该产品的信息，如下图所示：

MQTT_TSET	产品ID	用户ID	access_key ?	设备接入协议
空气监测仪 编辑 详情	366007	201572	查看	MQTTS

图 7.2.3 MQTT 产品信息

本实验会用到上述的产品信息，例如产品 ID（366007）、“access\_key”产品密钥以及产品名称（MQTT\_TSET）。

第三步：在设备列表中添加设备，如下图所示：

图 7.2.4 填写设备相关信息

这里也是需要用户自己填写设备的名称。填写完成之后可得到设备信息，如下图所示：

图 7.2.5 设备的 ID，设备名称和密钥 KEY

本实验会用到上图中的设备 ID（617747917）、设备名称 MQTT 以及“key”设备的密钥。

打开 OneNET 在线开发指南，在这个指南中找到服务器地址，这些服务器地址就是 MQTT 服务器地址，如下图所示：

连接协议	证书	地址	端口	说明
MQTT	<a href="#">证书下载</a>	mqttstls.heclouds.com	8883	加密接口
MQTT	-	mqttts.heclouds.com	1883	非加密接口

图 7.2.6 连接 OneNET 服务器的 IP 与端口

可以看到，OneNET 的 MQTT 服务器具有两个连接方式，一种是加密接口连接，而另一种是非加密接口连接，本章实验使用的是非加密接口连接 MQTT 服务器。

注意：MQTT 物联网套件采用安全鉴权策略进行访问认证，即通过核心密钥计算的 token 进行访问认证，简单来讲：若用户想连接 OneNET 的 MQTT 服务器，则必须计算核心密钥。这个核心密钥是根据前面创建的产品和设备相关的信息计算得来的，密钥的计算方法可以使用 OneNET 提供的 token 生成工具计算，该软件在这个网址下载：<https://open.iot.10086.cn/doc/v5/de>

[velop/detail/242](#)。

下面笔者简单讲解一下 token 生成工具的使用，如下图所示：

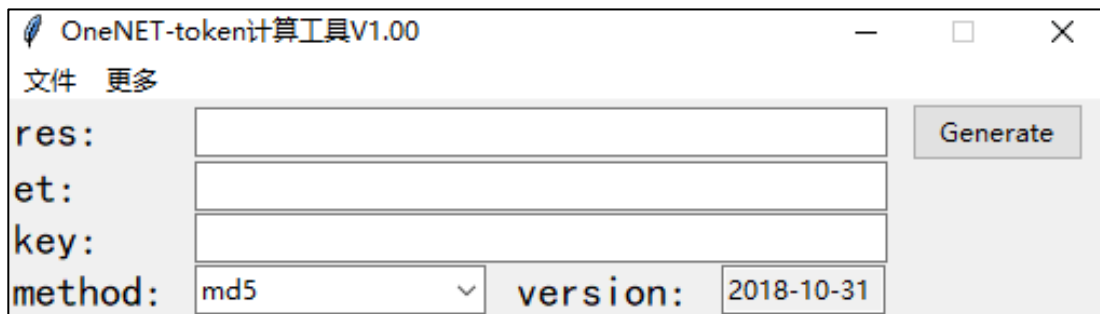


图 7.2.7 token 生成工具主界面

接下来笔者分别地讲解这个生成工具各个选项的作用，如下所示：

res: 输入格式为 “products/{pid}/devices/{device\_name}”，这个输入格式中的 “pid” 就是 MQTT 产品 ID，而 “device\_name” 就是设备的名称。根据前面创建的产品和设备来填写 res 选项的参数，如下图所示：

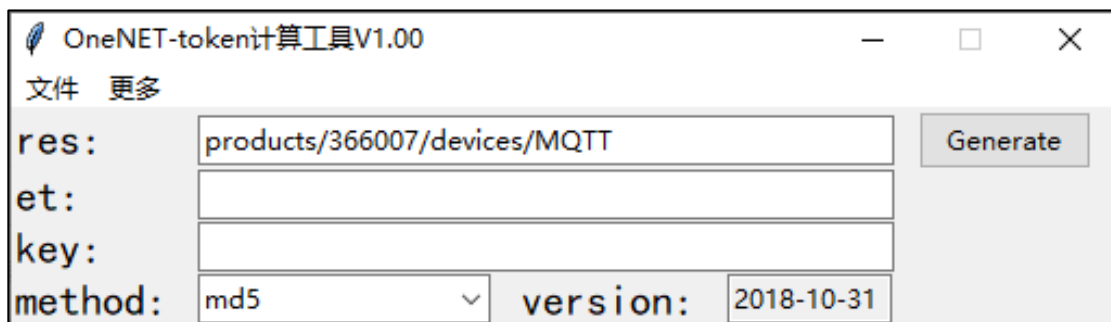


图 7.2.8 填写 res 配置项

et: 访问过期时间（expirationTime, unix）时间，设置访问时间应大于当前的时间，这里笔者选择参考文档中的数值（1672735919），如下图所示：

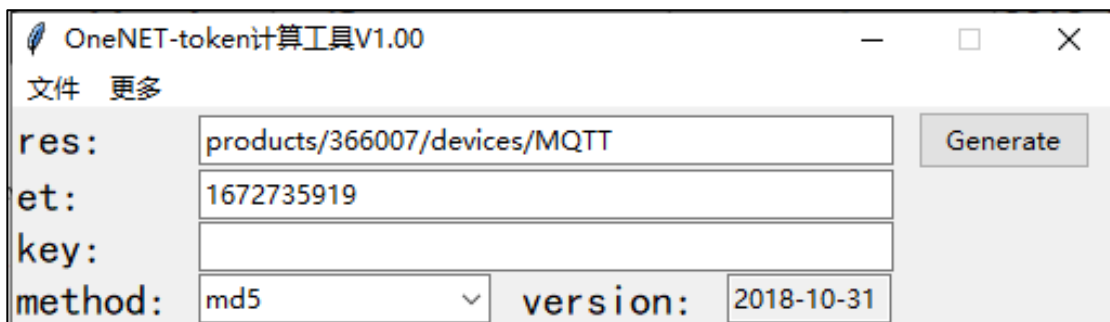


图 7.2.9 设置 et 配置项

key: 是指选择设备的 key 密钥，如下图所示：



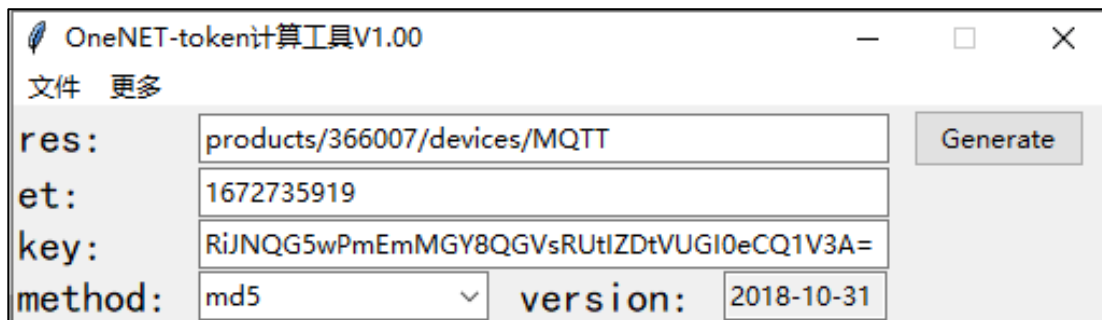


图 7.2.10 设置 key 配置项

最后按下上图中的“Generate”按键生成核心密钥，如下图所示。

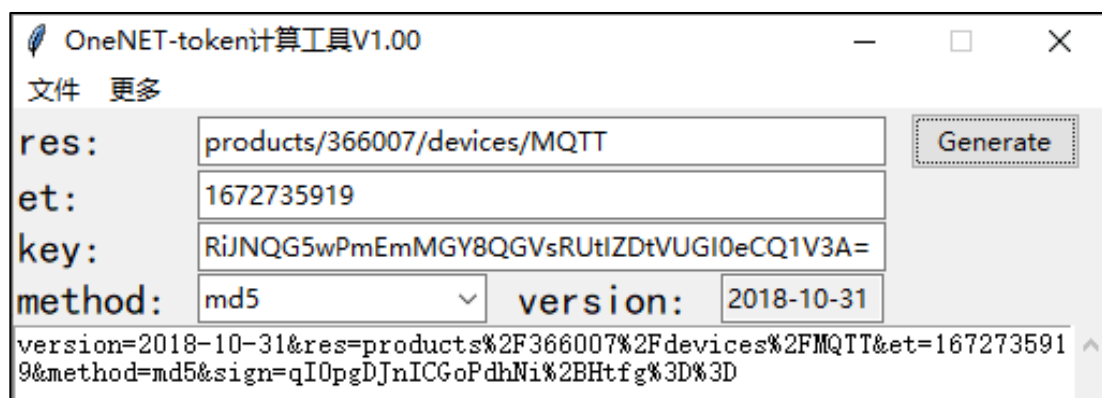


图 7.2.11 生成核心密钥

这个核心密钥会在 MQTT 客户端的结构体 client\_pass 成员变量保存。

### 7.3 工程配置

小上节我们使用 token 生成工具根据产品信息以及设备信息来计算核心密钥，这样的方式导致每次创建一个设备都必须根据这个设备信息再计算一次核心密钥才能连接，这种方式会大大地降低我们的开发效率。为了解决这个问题，笔者使用另一个方法，那就是使用代码的方式计算核心密钥。这些代码怎么编写呢？其实读者可在 OneOS 官网下载 OneOS 源码包，它里面包含了 MQTT 协议连接 OneNET 平台的核心密钥计算代码，这些代码在 oneos2.0\components\cloud\onenet\mqtt-kit\authorization 路径下找到，大家先下载 OneOS 源码并在该路径下复制 token 文件夹到工程的 User\APP 路径当中，如下如图所示：

名称	类型	大小
token	文件夹	
ch395_demo.c	C 文件	12 KB
ch395_demo.h	C/C++ Header	3 KB

图 7.3.1 保存 token 算法

打开工程并新建 User/APP/token 分组，在这个分组中添加 User\APP\token 路径下的.c 文件如下图所示：

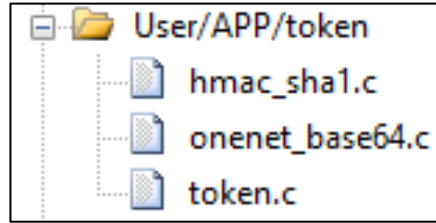


图 7.3.2 token 算法文件

## 7.4 基于 OneNET 平台 MQTT 实验

### 7.4.1 硬件设计

#### 1. 例程功能

本章目标是开发板使用 MQTT 协议连接 OneNET 服务器，并实现数据上存及更新。

该实验的实验工程，请参考《网络实验 7 CH395\_OneNET\_MQTT 实验》。

### 7.4.2 软件设计

#### 7.4.2.1 MQTT 函数解析

本实验是在《网络实验 3 CH395\_TCP 客户端实验》实验的基础上修改的，这里笔者重点讲解一下 ch395\_demo.c 文件下的函数，如下所示：

##### 1. 函数 ch395\_demo

测试通讯、SPI 初始化、网络参数初始化以及打开 socket 以及 MQTT 连接、订阅及发布等操作，该函数的原型如下所示：

```
void ch395_demo(void)
```

函数形参：

无。

返回值：

无。

##### 2. 函数 ch395\_show\_mesg

显示实验信息，该函数原型如下所示：

```
void ch395_show_mesg(void)
```

函数形参：

无。

返回值：

无。

##### 3. 函数 ch395\_transport\_send\_packet\_buffer

发送数据到 OneNET 服务器，该函数原型如下所示：

```
int ch395_transport_send_packet_buffer( int sock,
                                         unsigned char* buf,
                                         int buflen)
```

函数形参：

函数形参	描述
sock	sock 通道
buf	buf 数据首地址
buflen	buflen 数据长度

表 7.4.2.1.1 ch395\_transport\_send\_packet\_buffer 函数形参描述

**返回值：**

大于 0 表示发送成功。

#### 4. 函数 ch395\_transport\_get\_data

获取返回的数据，该函数原型如下所示：

```
int ch395_transport_get_data(unsigned char *buf, int count)
```

**函数形参：**

函数形参	描述
buf	buf 数据首地址
count	count 数据长度

表 7.4.2.1.1 ch395\_transport\_get\_data 函数形参描述

**返回值：**

大于 0 表示发送成功。

### 7.4.2.2 MQTT 配置步骤

#### ① 配置 CH395 为 TCP 客户端模式

CH395Q 配置为 TCP 客户端方式，请参考第三章的内容。

#### ② 设置三元组内容

在 OneNET 平台创建 MQTT 服务器，创建完成之后得到产品 ID、产品密钥、设备 ID 和设备密钥等参数。

#### ③ 定义发布和订阅命令

根据 OneNET 平台的要求定义发布和订阅命令。

### 7.4.2.3 程序流程图

本实验的程序流程图，如下图所示：

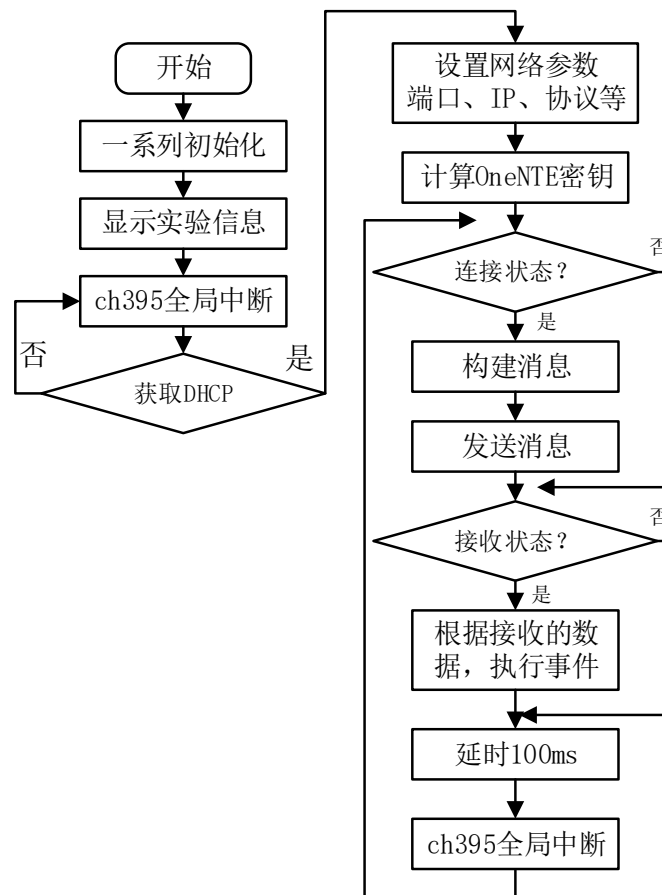


图 7.4.2.3.1 MQTT 实验流程图

#### 7.4.2.4 程序解析

打开 ch395\_demo.h 文件，在这个文件中笔者定义了 OneNET 服务器连接的产品 ID 与设备 APIKey 等参数，另外该文件还声明了 ch395\_demo.c 下的函数，接下来笔者分别地讲解这两个文件实现代码，如下所示：

##### 1. 文件 ch395\_onenet.h

```

#ifndef __CH395_DEMO_H
#define __CH395_DEMO_H
#include "../SYSTEM/sys/sys.h"

/* 用户需要根据设备信息完善以下宏定义中的三元组内容 */
#define USER_DEVICE_NAME "MQTT" /* 设备名 */
#define USER_PRODUCT_ID "366007" /* 产品 ID */
/* 产品密钥 */
#define USER_ACCESS_KEY "qlWudWg/3ANGVQLeHGfAu0Eh8J7CWgozfOpljI+Gy8k="
#define USER_DEVICE_ID "617747917" /* 产品设备 ID */
/* 设备密钥 */
#define USER_KEY "QyxIRiJNQG5wPmEmMGY8QGVsRUtIZDtVUGI0eCQ1V3A="

```

```

/* 该密码需要 onenet 提供的 token 软件计算得出 */
#define PASSWORD "version=2018-10-
31&res=products%2F366007%2Fdevices%2FMQTT&et=1672735919&method=md5&sign=qI0pgDJ
nICGoPdhNi%2BHtfg%3D%3D"

/* 以下参数的宏定义固定，不需要修改，只修改上方的参数即可 */
#define HOST_NAME "open.iot.10086.cn" /* onenet 域名 */
#define DEVICE_SUBSCRIBE "$sys/"USER_PRODUCT_ID"/"USER_DEVICE_NAME"/dp/post/json/" /* 订阅 */
#define DEVICE_PUBLISH "$sys/"USER_PRODUCT_ID"/"USER_DEVICE_NAME"/dp/post/json" /* 发布 */
#define SERVER_PUBLISH "$sys/"USER_PRODUCT_ID"/"USER_DEVICE_NAME"/cmd/request/" /* 服务器下发命令 */

typedef struct
{
    char pro_id[10];
    char access_key[48];
    char dev_name[64 + 1];
    char dev_id[16];
    char key[48];
} onenet_info_t;

void ch395_demo(void); /* 例程测试 */

#endif

```

上述的订阅和发布的 topic 主题，请大家参考 OneNET 官方文档，该文档的地址为 <https://open.iot.10086.cn/doc/v5/develop/detail/251>，接着笔者定义了 onenet\_info\_t 结构体，该结构体主要存储 MQTT 产品和设备信息，最后笔者在此文件声明了 ch395\_demo 函数，提供外部文件使用。

## 2. 文件 ch395\_demo.c

此文件笔者重点讲解 ch395\_demo 函数，该函数分别完成了四个任务，下面笔者分别地讲解这四个任务的内容。

### ① 配置 CH395Q 为客户端模式

```

cha95_sockct_sta[0].socket_enable = CH395Q_ENABLE; /* 使能 socket 接口 */
cha95_sockct_sta[0].socket_index = CH395Q_SOCKET_0; /* 设置 socket 接口 */
/* 设置目标 IP 地址 */
memcpy(ch395_sockct_sta[0].des_ip, ch395_des_ipaddr,
        sizeof(ch395_sockct_sta[0].des_ip));
/* 设置静态本地 IP 地址 */
memcpy(ch395_sockct_sta[0].net_config.ipaddr, ch395_ipaddr,
        sizeof(ch395_sockct_sta[0].net_config.ipaddr));
/* 设置静态网关 IP 地址 */
memcpy(ch395_sockct_sta[0].net_config.gwipaddr, ch395_gw_ipaddr,

```

```

        sizeof(cha95_sockct_sta[0].net_config.gwipaddr));
/* 设置静态子网掩码地址 */
memcpy(cha95_sockct_sta[0].net_config.maskaddr, ch395_ipmask,
        sizeof(cha95_sockct_sta[0].net_config.maskaddr));
/* 设置静态 MAC 地址 */
memcpy(cha95_sockct_sta[0].net_config.macaddr, ch395_macaddr,
        sizeof(cha95_sockct_sta[0].net_config.macaddr));
cha95_sockct_sta[0].des_port = 1883; /* 目标端口 */
cha95_sockct_sta[0].sour_port = 5000; /* 源端口 */
cha95_sockct_sta[0].proto = CH395Q_SOCKET_TCP_CLIENT; /* 设置协议 */
cha95_sockct_sta[0].send.buf = socket0_send_buf; /* 发送数据 */
cha95_sockct_sta[0].send.size = sizeof(socket0_send_buf); /* 发送数据大小 */
cha95_sockct_sta[0].recv.buf = socket0_recv_buf; /* 接收数据缓冲区 */
cha95_sockct_sta[0].recv.size = sizeof(socket0_recv_buf); /* 接收数据大小 */
ch395q_socket_config(&cha95_sockct_sta[0]); /* 配置 socket 参数 */

```

配置 CH395Q 芯片为 TCPClient 模式，并设置目标端口为 1883。

## ② 计算连接核心密钥

这里我们根据前面 7.3 小节的内容来计算 OneNET 服务器核心密钥，如下源码所示：

```

char pro_id[] = USER_PRODUCT_ID; /* 产品 ID */
char access_key[] = USER_ACCESS_KEY; /* 产品密钥 */
char dev_name[] = USER_DEVICE_NAME; /* 设备名称 */
char dev_id[] = USER_DEVICE_ID; /* 产品设备 ID */
char key[] = USER_KEY; /* 设备密钥 */

char version[] = "2018-10-31";
unsigned int expiration_time = 1956499200;
char authorization_buf[160] = {0};

/* 把各个参数保存在 g_onenet_info 结构体的成员变量中 */
memset(g_onenet_info.pro_id, 0, sizeof(g_onenet_info.pro_id));
strcpy(g_onenet_info.pro_id, pro_id);

memset(g_onenet_info.access_key, 0, sizeof(g_onenet_info.access_key));
strcpy(g_onenet_info.access_key, access_key);

memset(g_onenet_info.dev_name, 0, sizeof(g_onenet_info.dev_name));
strcpy(g_onenet_info.dev_name, dev_name);

memset(g_onenet_info.dev_id, 0, sizeof(g_onenet_info.dev_id));
strcpy(g_onenet_info.dev_id, dev_id);

memset(g_onenet_info.key, 0, sizeof(g_onenet_info.key));
strcpy(g_onenet_info.key, key);

```

```
/* 根据这些参数进行解码, 当然这个密码可以在 token 软件下解码 */
onenet_authorization(version,
    (char *)g_onenet_info.pro_id,
    expiration_time,
    (char *)g_onenet_info.key,
    (char *)g_onenet_info.dev_name,
    authorization_buf,
    sizeof(authorization_buf),
    0);

data.clientID.cstring = (char *)g_onenet_info.dev_name; /* 设备名称 */;
data.username.cstring = (char *)g_onenet_info.pro_id; /* 产品 ID */;
data.password.cstring = (char *)authorization_buf; /* 计算出来的密码 */;
data.keepAliveInterval = 100; /* 保活时间 */
```

笔者根据 OneNET 服务器创建的产品信息, 使用 onenet\_authorization 函数计算核心密钥, 并把它存储在 authorization\_buf 缓存区当中。

### ③ 连接 OneNET 服务器

根据上述的核心密钥和创建 MQTT 产品的信息连接 OneNET 平台, 如下源码所示:

```
case CONNECT: /* 客户端发送服务器的连接操作 */
    /* 获取数据组长发送连接信息 */
    g_len = MQTTSerialize_connect((unsigned char *)socket0_send_buf,
        sizeof(socket0_send_buf), &data);

    /* 发送返回发送数组长度 */
    g_rc = ch395_transport_send_packet_buffer(CH395Q_SOCKET_0,
        (unsigned char *)socket0_send_buf,
        g_len);

    if (g_rc == g_len)
        printf("发送连接成功\r\n");
    else
        printf("发送连接失败\r\n");

    g_msgtypes = 0;
    break;
```

上述源码调用了 MQTTSerialize\_connect 函数把 MQTT 相关的信息转换成序列码, 接着调用 ch395\_transport\_send\_packet\_buffer 发送连接序列码。

### ④ 订阅操作

发送连接序列码之后程序就会发送一个订阅主题到 OneNET 平台, 如下源码所示:

```
case SUBSCRIBE: /* 客户端发送到服务器的订阅操作 */
    topicString.cstring = DEVICE_SUBSCRIBE;
    g_len = MQTTSerialize_subscribe((unsigned char *)socket0_send_buf,
        sizeof(socket0_send_buf), 0, 1, 1, &topicString, &g_req_qos);
    g_rc = ch395_transport_send_packet_buffer(CH395Q_SOCKET_0,
```

```

        (unsigned char *)socket0_send_buf, g_len);

if (g_rc == g_len)
    printf("send SUBSCRIBE Successfully\r\n");
else
{
    int t = 0;

    t ++;

    if (t >= 10)
    {
        t = 0;
        g_msgtypes = CONNECT;
    }
    else
        g_msgtypes = SUBSCRIBE;
    break;
}
g_msgtypes = 0;
break;

```

可以看到，DEVICE\_SUBSCRIBE 配置项指向的是订阅主题字符串，接着程序调用函数 MQTTSerialize\_subscribe 把订阅主题数据转换成序列码，最后调用函数 ch395\_transport\_send\_packet\_buffer 发送该序列码，以表示订阅操作。

### ⑤ 发布数据

当订阅完成以后才能实施发布措施，如下源码所示：

```

if (g_ch395q_sta.switch_status == CONNECT_STAT)
{
    g_temp = 30 + rand() % 10 + 1;    /* 温度的数据 */
    g_humid = 54.8 + rand() % 10 + 1; /* 湿度的数据 */
    sprintf((char *)payload_out, "{\"id\": 123,\"dp\": { \"temperatrue\": [\"v\": %0.1f,}],\"power\": [{\"v\": %0.1f,}]}}", g_temp, g_humid);
    payload_out_len = strlen((char *)payload_out);
    topicString.cstring = DEVICE_PUBLISH; /* 属性上报 发布 */
    g_len = MQTTSerialize_publish((unsigned char *)socket0_send_buf,
        sizeof(socket0_send_buf), 0, 1, 0, 1, topicString, payload_out,
        payload_out_len);
    g_rc = ch395_transport_send_packet_buffer(CH395Q_SOCKET_0,
        (unsigned char *)socket0_send_buf, g_len);

    if (g_rc == g_len)
    {
        printf("send PUBLISH Successfully\r\n");
    }
}

```



```

    }
    else
    {
        printf("send PUBLISH failed\r\n");
    }
}

delay_ms(100);

```

从上述源码可以看出，payload\_out 存储的是发布的数据，这个数据结构必须符合 OneNET 平台的要求，接着 DEVICE\_PUBLISH 配置项指向发布操作的指令，它们经过函数 MQTTSerialize\_publish 转换成序列码，接着调用函数 ch395\_transport\_send\_packet\_buffer 发送序列码，这样才能发布成功。

### 7.4.3 下载验证

编译代码，并把下载到开发板上运行，打开 OneNET 的 MQTT 服务器查看数据流展示，如下图所示：

temperature 2020-08-07 17:53:41	...	humidity 2020-08-07 17:53:41
27		82

图 7.4.3.1 温湿度显示成功

## 第八章 原子云平台连接

原子云即原子云服务器，是正点原子推出的物联网云服务平台，目前它可以实现数据的监控、转发和管理等功能，在未来也会持续更新更多的功能以满足用户的需求。原子云域名为：[cloud.alientek.com](http://cloud.alientek.com)，端口号为：59666。原子云已经支持通过 API 接口进行访问，相关文档说明在“资料包 → 1，文档资料”下查找。

本章我们分为几个部分讲解：

### 8.1 原子云工作流程

#### 8.1 原子云连接实验

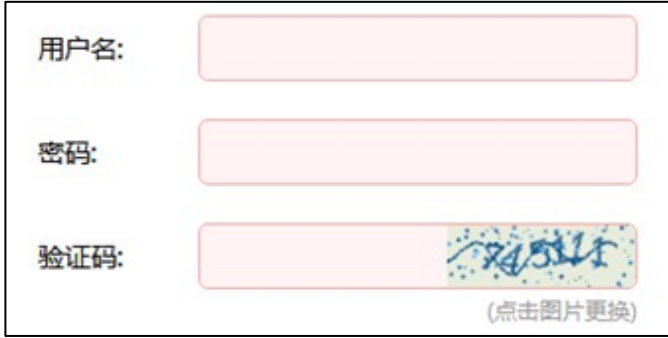
### 8.1 原子云工作流程

#### 如何获取设备编号与设备密码

本小节就来教大家如何从原子云上获取设备编号与设备密码,以实现开发板与原子云之间的数据透传，用户不仅可以在原子云上查看设备上传到原子云的数据，而且可以使用 API 接入原子云来获取这些数据，以便用户开发。

原子云获取设备编号与设备密码步骤：

**第一步：账号注册。**登陆原子云服务器：<https://cloud.alientek.com/>，没有账号可以先注册一个账号，有了账号后直接输入用户名和密码登录原子云就可以了。如下图所示：



该图展示了原子云的登录界面。界面包含三个输入框，分别用于输入“用户名”、“密码”和“验证码”。验证码框右侧有一个包含随机数字和字母的图形验证码，下方有“(点击图片更换)”的提示文字。

图 8.1.1.1 登陆原子云界面

**第二步：创建设备节点。**进入“设备管理”界面，点击“新增设备”开始创建设备节点，在“新增设备/选择设备类型”中选择“ATK-UART2ETH”类型，然后输入“设备名称和密码”，选择“新增”就可以成功创建一个设备。创建成功的设备节点会在列表中显示。如下图所示：

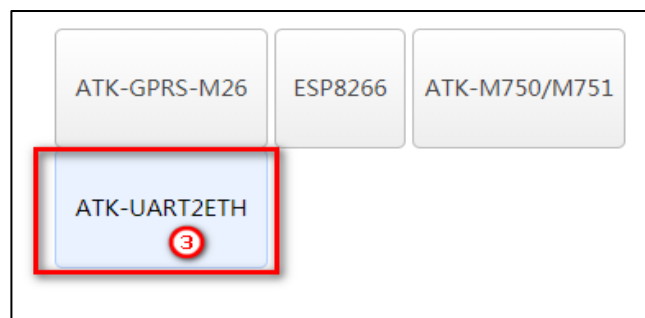


图 8.1.1.2 新增设备节点

**第三步：配置 DTU 的设备编号与设备密码参数。**新增设备节点成功后，就能在设备节点

列表中查看刚刚新增成功的设备节点，将上一步中新增的设备编号和设备密码填入函数 `atk_decode` 中就可以实现开发板与原子云之间的连接，如下图所示：

```
/* 以下参数的宏定义固定，不需要修改，只修改上方的参数即可 */
#define HOST_NAME "cloud.alientek.com" /*原子云域名*/
#define HOST_PORT 59666 /*原子云端口号*/
#define LWIP_SEND_DATA 0X80 /*定义有数据发送*/
extern uint8_t tcp_client_flag; /*TCP客户端数据发送标志位*/
#define DEVICE "61259083526781695524" /*设备号*/
#define PAW "12345678" /*设备密码*/
```

图 8.1.1.3 原子云参数配置相关

**第四步：在分组中添加 lib 文件。**在工程添加 `atk_decobe.lib` 和 `atk.h` 文件，如下图所示：

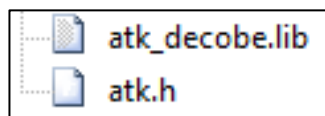


图 8.1.1.4 添加原子云的加密文件

## 8.2 原子云连接实验

### 8.2.1 硬件设计

#### 1. 例程功能

实现开发板与正点原子的物联网云服务平台连接与交互数据。当按下 KEY0 时，把数据上存到原子云平台上，当然原子云平台下发数据可在串口上打印。

该实验的实验工程，请参考《网络实验 8 CH395\_连接原子云》。

### 8.2.2 软件设计

#### 8.2.2.1 原子云函数解析

##### `atk_decode` 函数

该函数是实现 MQTT 连接、收发等功能，该函数的原型如以下源码所示：

```
uint8_t * atk_decode(char *mpid, char *pwd);
```

##### 函数形参：

函数 `atk_decode()` 具有 2 个参数，如表 9.2.2.1.1 所示：

参数	描述
<code>mpid</code>	设备号
<code>pwd</code>	设备密码

表 8.2.2.1.1 `atk_decode()` 形参描述

##### 函数返回值：

无。

#### 8.2.2.2 原子云配置步骤

##### ① 配置 CH395 为 TCP 客户端模式

CH395Q 配置为 TCP 客户端方式，请参考第三章的内容。

##### ② 配置端口号

配置目标端口和源端口为原子云端口 59666。

### ③ 计算连接密钥

把设备号和设备密码传入 `atk_decode` 函数来计算连接密钥。

### ④ 连接原子云

把计算得出的密钥以 `ch395_send_data` 函数发送。

## 8.2.2.3 程序流程图

本实验的程序流程图，如下图所示：

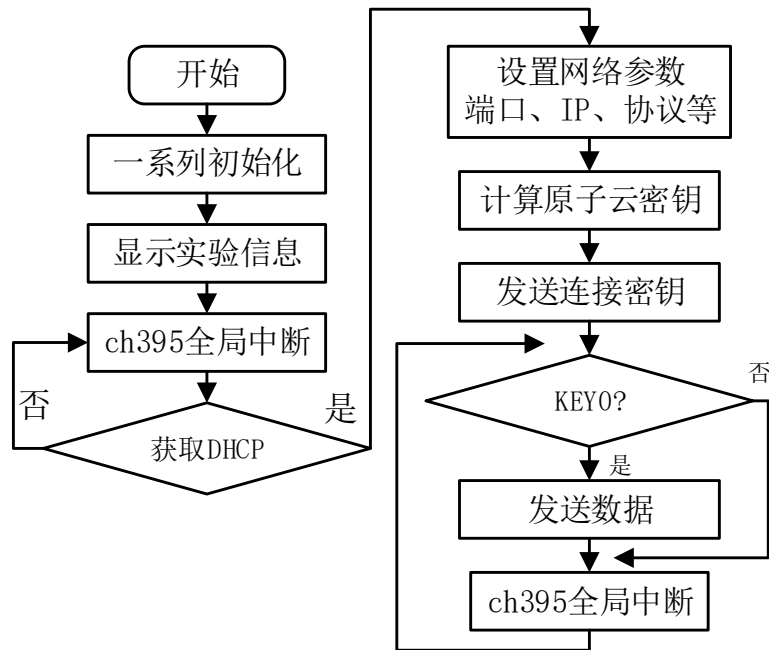


图 8.2.2.3.1 连接原子云实验

## 8.2.2.4 程序解析

笔者重点讲解 `ch395_demo.c` 文件的实现代码，如下所示：

### 1. `ch395_demo.c` 文件

该文件定义了 2 个函数，如下表所示：

函数	描述
<code>ch395_show_mesg()</code>	显示实验信息
<code>ch395_demo()</code>	初始化、TCP 连接以及原子云连接操作

表 8.2.2.4.1 `ch395_demo.c` 文件下的函数描述

### ① 定义网络参数及连接参数

```

/* 本地网络信息：IP 地址、网关地址、子网掩码和 MAC 地址 */
uint8_t ch395_ipaddr[4]      = {192,168,1,10};
uint8_t ch395_gw_ipaddr[4]  = {192,168,1,1};
uint8_t ch395_ipmask[4]     = {255,255,255,0};
uint8_t ch395_macaddr[6]    = {0xB8,0xAE,0x1D,0x00,0x00,0x00};

/* 远程 IP 地址设置 */
uint8_t ch395_des_ipaddr[4] = {47, 98, 186, 15};
  
```

```
static uint8_t socket0_send_buf[] = {"This is from CH395Q\r\n"};
static uint8_t socket0_recv_buf[1024];
ch395_socket cha95_sockct_sta[8];
```

```
#define DEVICE "33057282794714357363" /* 设备号 */
#define PAW "12345678" /* 设备密码 */
```

ch395\_des\_ipaddr 数组为原子云的 IP 地址，DEVICE 和 PAW 为设备的 ID 和密钥。

### ② 显示实验信息

```
/**
 * @brief      显示实验信息
 * @param      无
 * @retval     无
 */
void ch395_show_mesg(void)
{
    /* LCD 显示实验信息 */
    lcd_show_string(10, 10, 220, 32, 32, "STM32", RED);
    lcd_show_string(10, 47, 220, 24, 24, "CH395Q YuanZiYun", RED);
    lcd_show_string(10, 76, 220, 16, 16, "ATOM@ALIENTEK", RED);
    lcd_show_string(10, 97, 200, 16, 16, "KEY0: Send", BLUE);

    /* 串口输出实验信息 */
    printf("\n");
    printf("*****\r\n");
    printf("STM32\r\n");
    printf("CH395Q YuanZiYun\r\n");
    printf("ATOM@ALIENTEK\r\n");
    printf("KEY0: Send\r\n");
    printf("*****\r\n");
    printf("\r\n");
}
```

显示实验信息和串口打印实验信息。

### ③ 测试实验

```
/**
 * @brief      例程测试
 * @param      无
 * @retval     无
 */
void ch395_demo(void)
{
    uint8_t key = 0;

    ch395_show_mesg(); /* 显示信息 */
```

```

do
{
    ch395q_handler();
}

while (g_ch395q_sta.dhcp_status == DHCP_STA); /* 获 DHCP*/
/* 使能 socket 接口 */
cha95_sockct_sta[0].socket_enable = CH395Q_ENABLE;
/* 设置 socket 接口 */
cha95_sockct_sta[0].socket_index = CH395Q_SOCKET_0;
/* 设置目标 IP 地址 */
memcpy(ch395_sockct_sta[0].des_ip, ch395_des_ipaddr,
        sizeof(ch395_sockct_sta[0].des_ip));
/* 设置静态本地 IP 地址 */
memcpy(ch395_sockct_sta[0].net_config.ipaddr, ch395_ipaddr,
        sizeof(ch395_sockct_sta[0].net_config.ipaddr));
/* 设置静态网关 IP 地址 */
memcpy(ch395_sockct_sta[0].net_config.gwipaddr, ch395_gw_ipaddr,
        sizeof(ch395_sockct_sta[0].net_config.gwipaddr));
/* 设置静态子网掩码地址 */
memcpy(ch395_sockct_sta[0].net_config.maskaddr, ch395_ipmask,
        sizeof(ch395_sockct_sta[0].net_config.maskaddr));
/* 设置静态 MAC 地址 */
memcpy(ch395_sockct_sta[0].net_config.macaddr, ch395_macaddr,
        sizeof(ch395_sockct_sta[0].net_config.macaddr));
/* 目标端口 */
cha95_sockct_sta[0].des_port = 59666;
/* 源端口 */
cha95_sockct_sta[0].sour_port = 59666;
/* 设置协议 */
cha95_sockct_sta[0].proto = CH395Q_SOCKET_TCP_CLIENT;
/* 发送数据 */
cha95_sockct_sta[0].send.buf = socket0_send_buf;
/* 发送数据大小 */
cha95_sockct_sta[0].send.size = sizeof(socket0_send_buf);
/* 接收数据缓冲区 */
cha95_sockct_sta[0].recv.buf = socket0_recv_buf;
/* 接收数据大小 */
cha95_sockct_sta[0].recv.size = sizeof(socket0_recv_buf);
/* 配置 socket 参数 */
ch395q_socket_config(&cha95_sockct_sta[0]);

ch395_send_data(CH395Q_SOCKET_0, (uint8_t *)atk_decode(DEVICE, PAW),

```

```

        strlen((char *)atk_decode(DEVICE, PAW)));

while (1)
{
    key = key_scan(0);

    if (key == KEY0_PRES)
    {
        ch395_send_data(CH395Q_SOCKET_0, (uint8_t *)socket0_send_buf,
                        strlen((char *)socket0_send_buf));
    }
    ch395q_handler();
}
}

```

从上述源码可知看出，根据设备号和设备密码计算连接密钥，并把连接密钥调用函数 `ch395_send_data` 发送到原子云服务器，以表示握手连接，当按下 `KEY0_PRES` 可向原子云发送数据。

### 8.2.3 下载验证

编译程序并下载到开发板上，打开原子云平台：<https://cloud.alientek.com/>，如下图所示：

序号	类型	名称	编号
1001	ATK-UART2ETH	节点1001	61259083526781695524

图 8.2.3.1 原子云平台主界面

按下开发板上的 `KEY0` 把 “This is from CH395Q\r\n” 数据发送到原子云服务器当中，原子云平台发送的数据可在串口上显示。