EAP平台源码分析

作者：滑国青

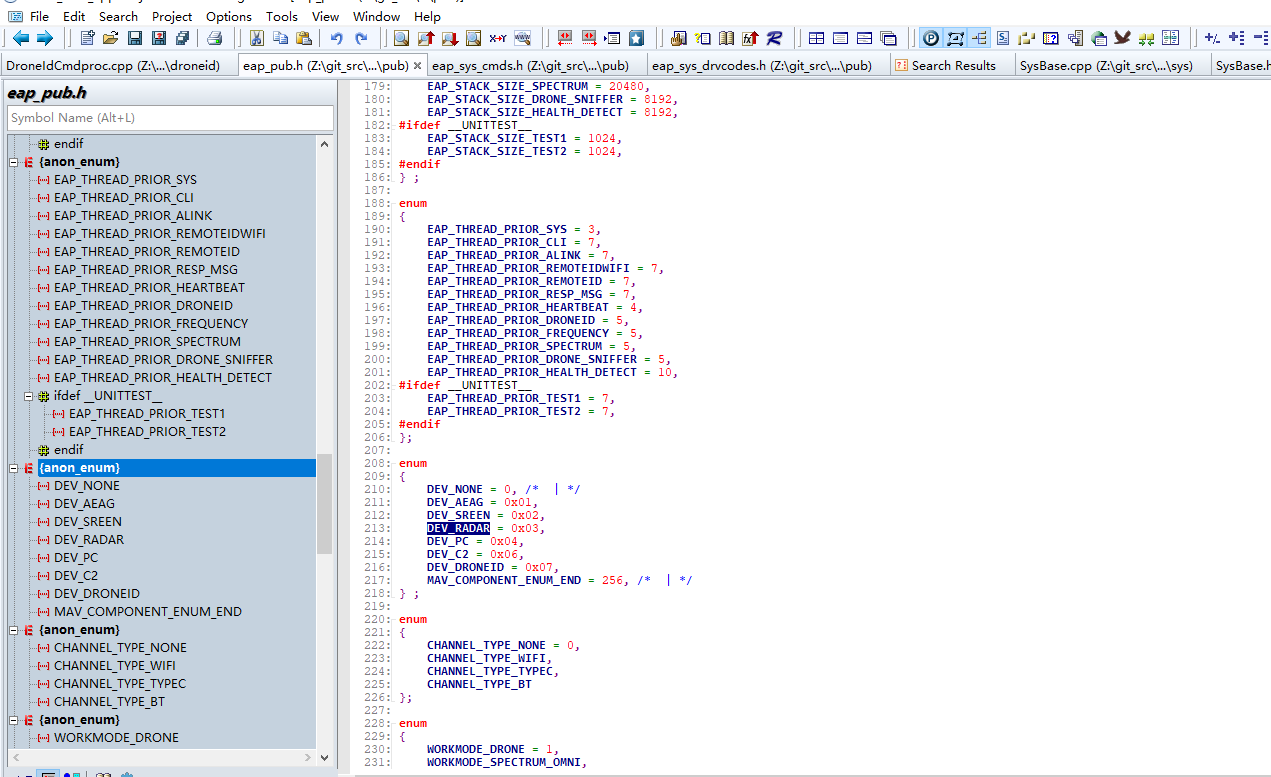
历史：2023.08.17 未完成，写作中

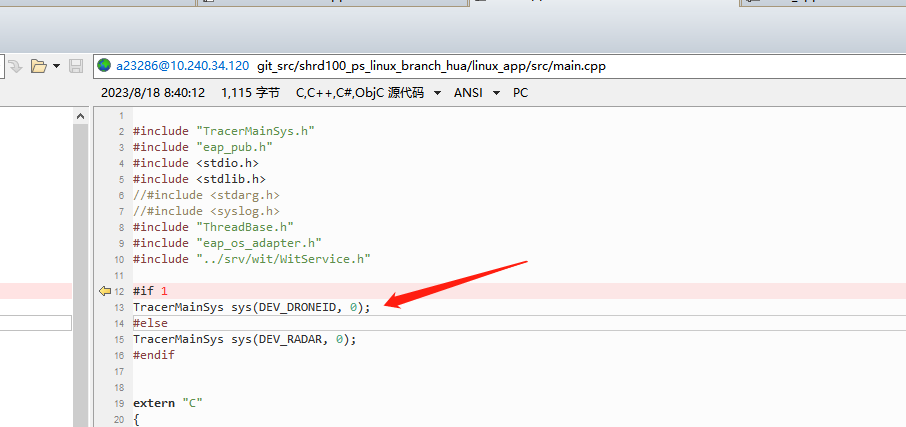
# 前言

# 重要宏定义

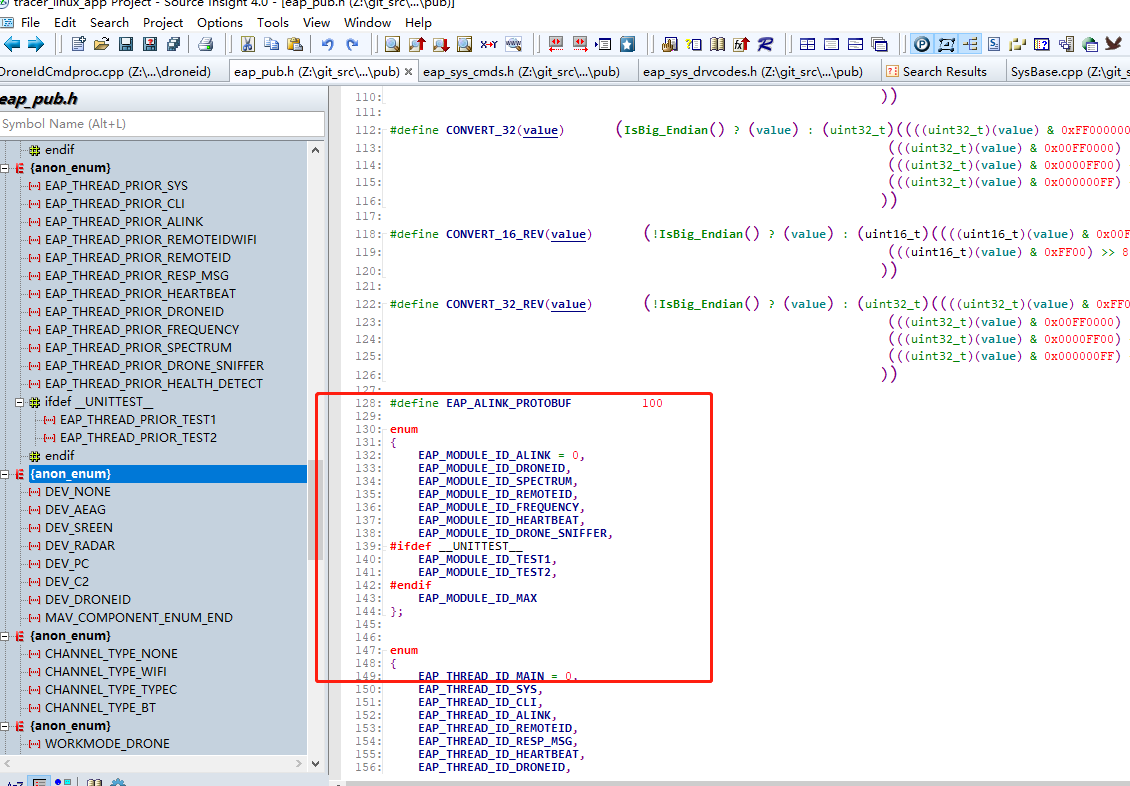
## 设备类型

eap\_pub.sh



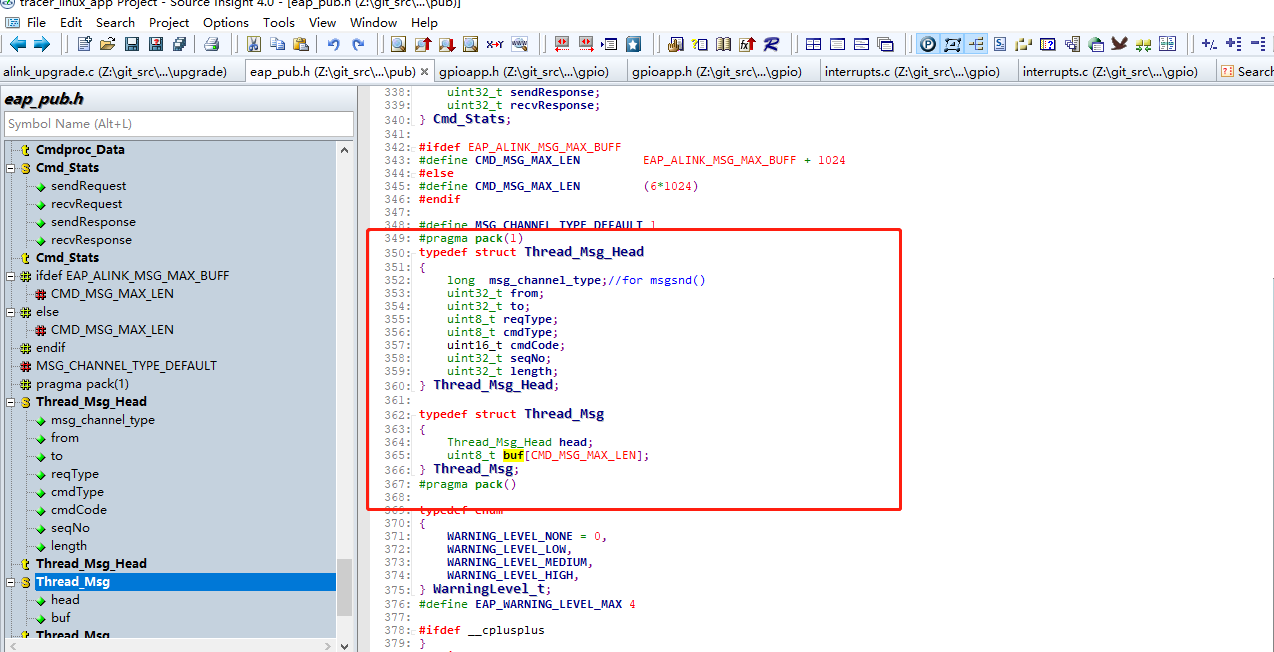


## 模块ID定义



## 模块间消息

### Thread\_Msg

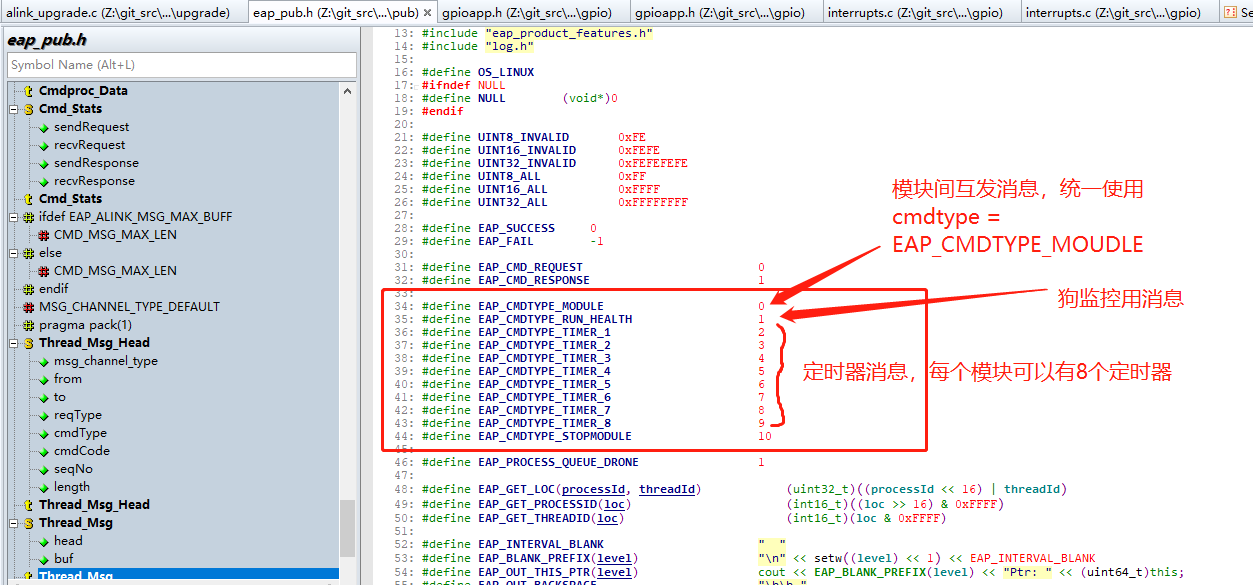


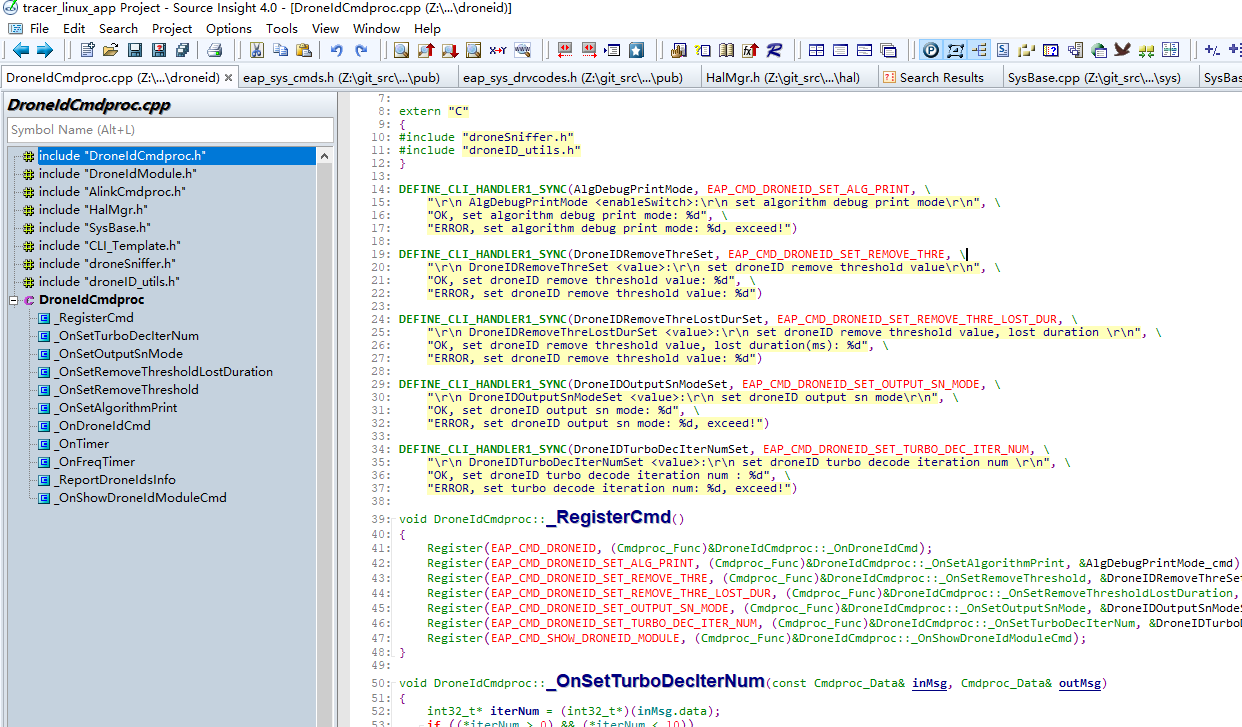
Msg\_channel\_type 给sysv queue msg收发使用，因为它的消息头是一个long类型的msgtype,所以如果消息采用sysv queue msg格式，必须预留此位置。

### 消息基本类型cmdtype

Cmdtype 这是消息基本类型。

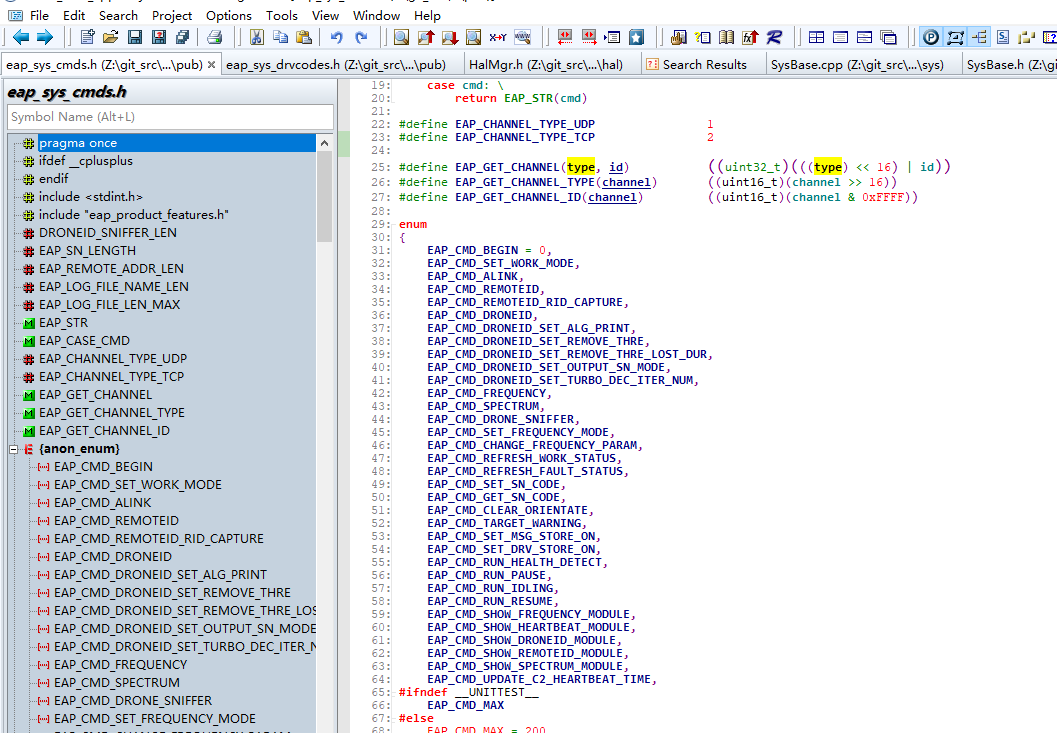
Cmdcode 这是一种消息内部的子类型





### 模块间消息子类型

eap\_sys\_cmds.h



### 发消息有两个基本接口



### SendAsynMsg与SendAsynData区别

sendAsyncMsg提供的buf是以Thread\_Msg为头的。

而sendAsyncData则是没有ThreadMsg头，由此函数加头后再调SendAsyncMsg发送出去。

所以SendAsyncMsg是最基本的发送函数。

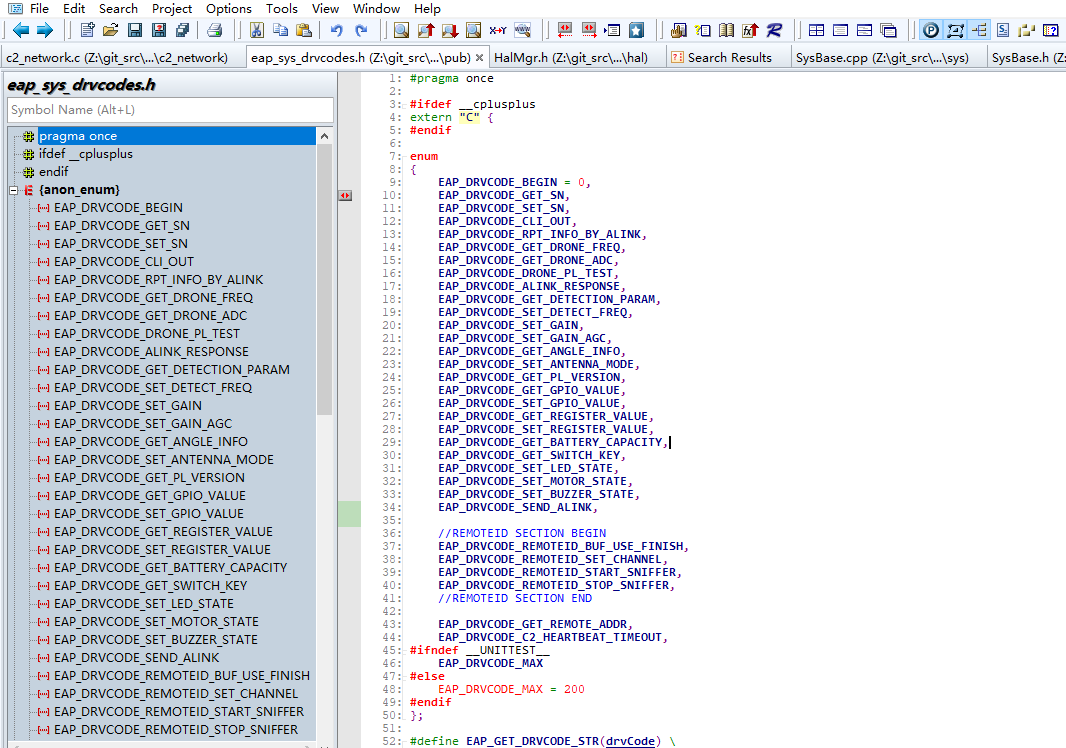
void SendAsynMsg(uint16\_t cmdCode, uint8\_t \*buf, uint32\_t dataLen)

void SendAsynData(uint16\_t cmdCode, uint8\_t \*data, uint32\_t dataLen)

## ALINK消息类型定义

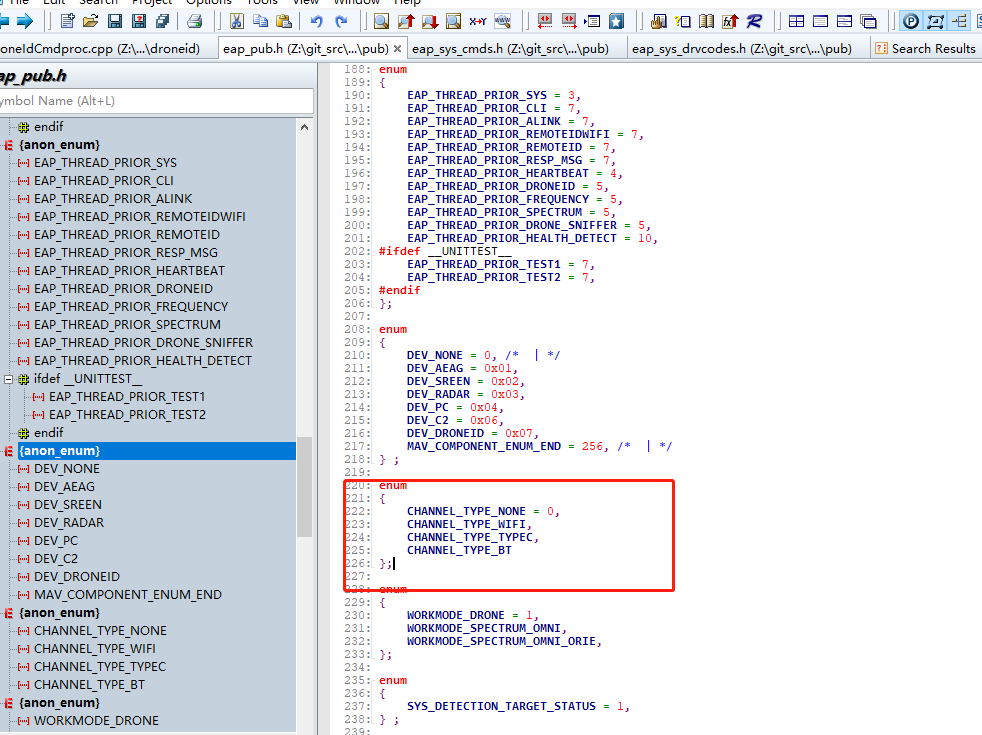
## HAL命令消息类型定义

eap\_sys\_drvcodes.h



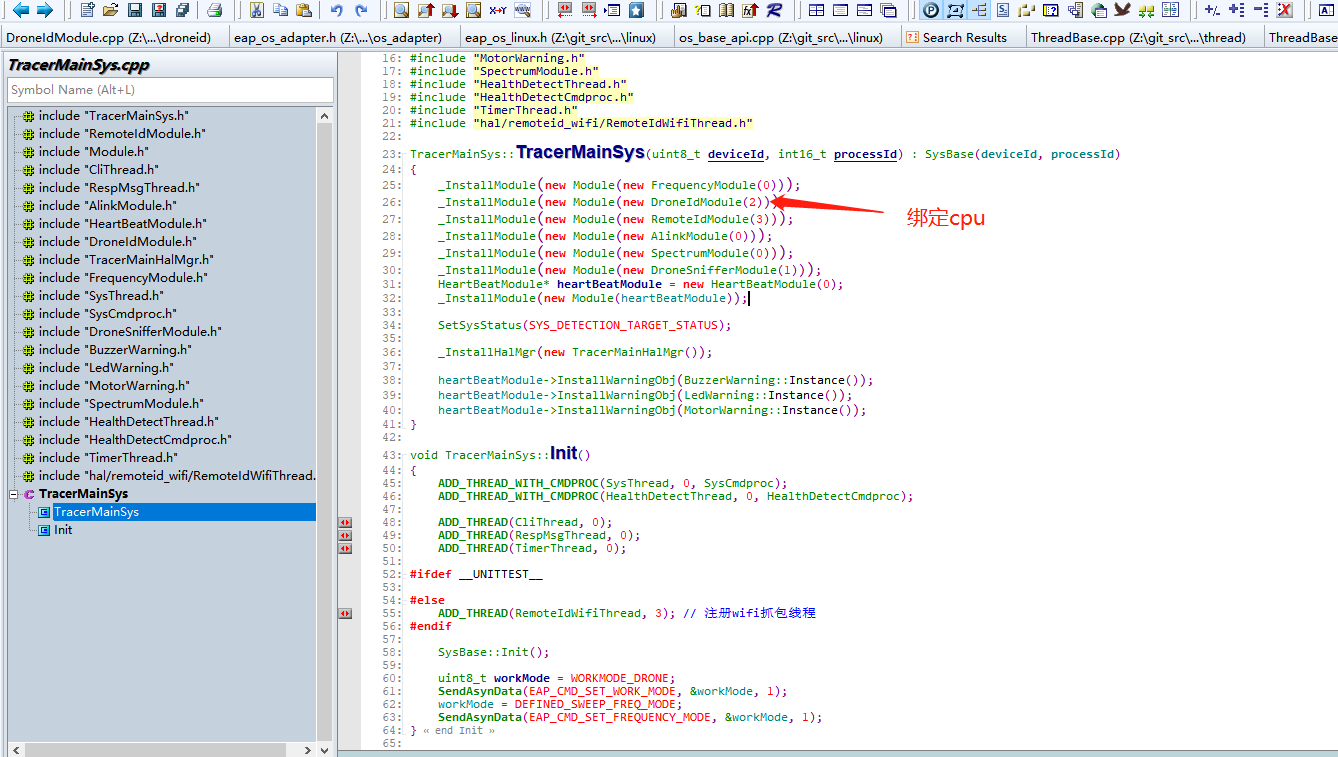
## 组网类型

eap\_pub.h



## 线程的Cpu绑定

void eap\_os\_set\_thread\_affinity(unsigned long int thread\_handle,uint16\_t core\_id)

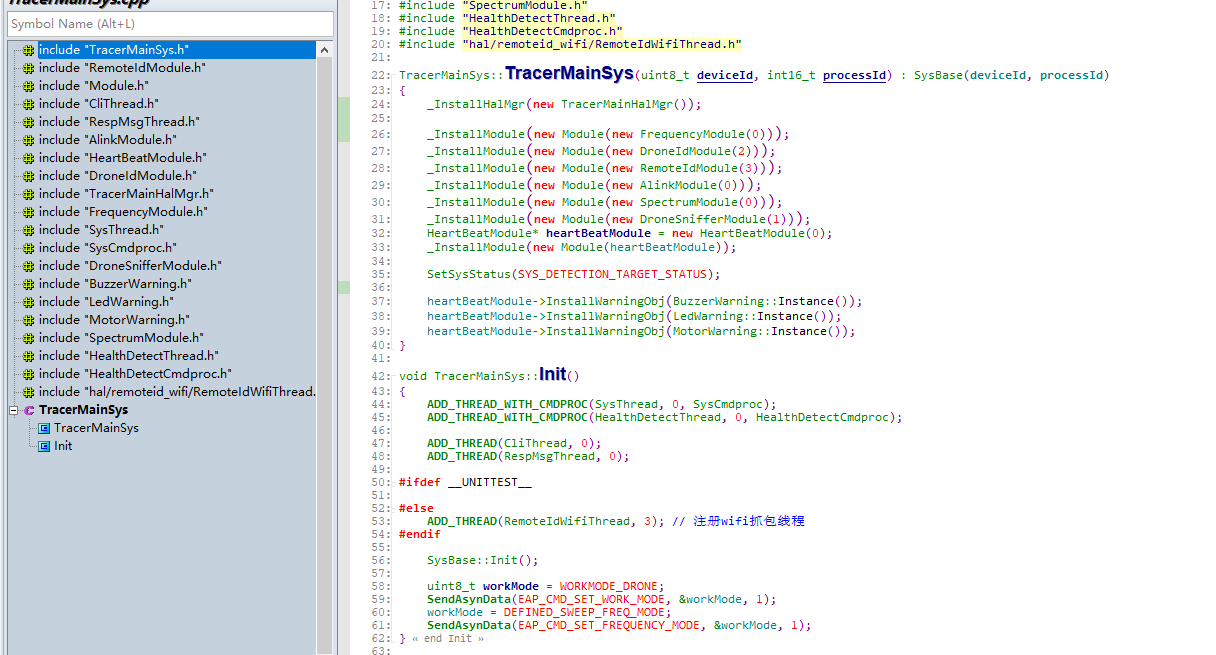


# 初始化

## class TracerMainSys

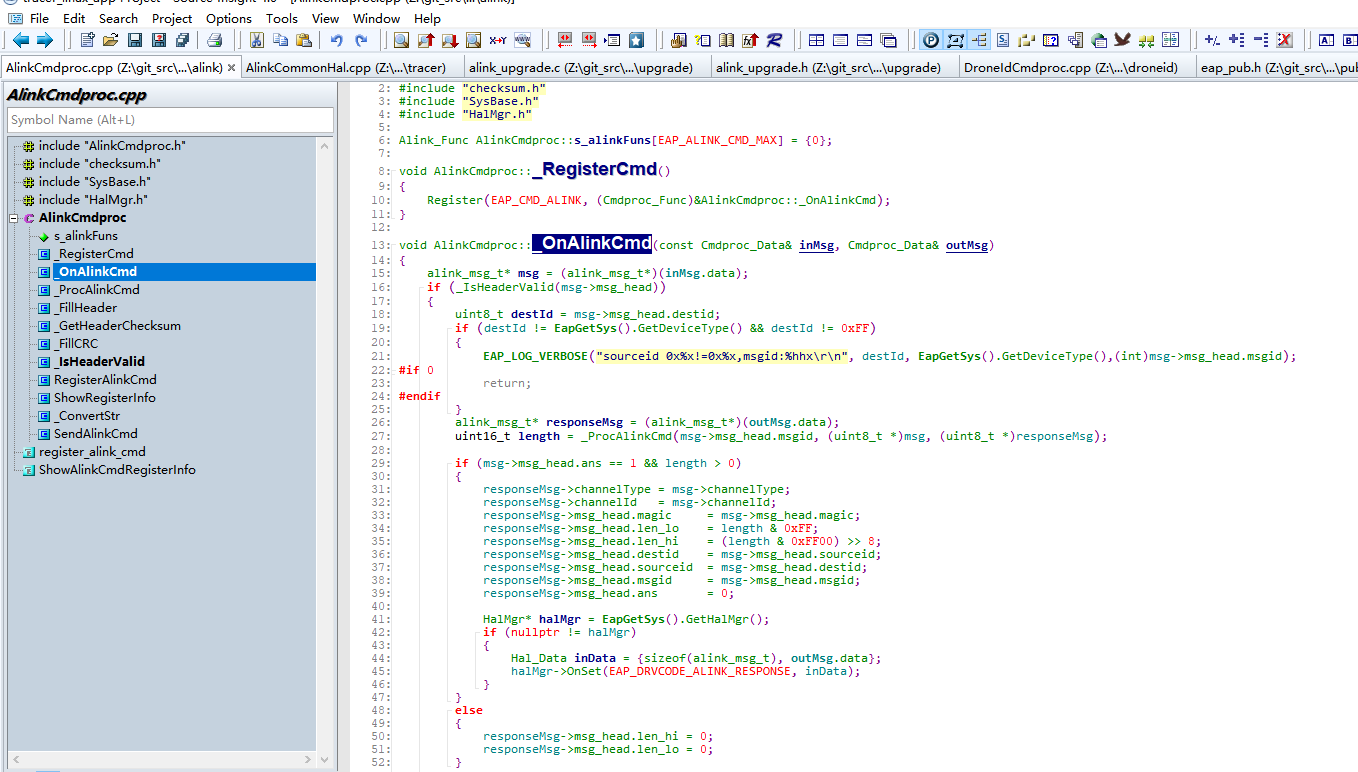
基类为SysBase

完成modules,hal及一些专用服务线程的初始化

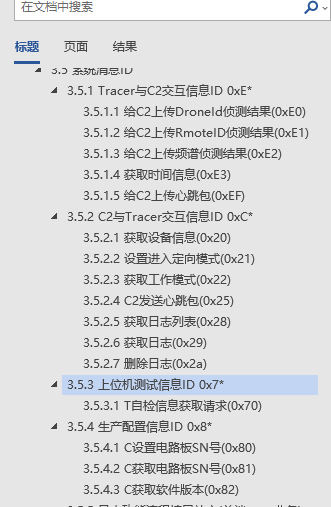


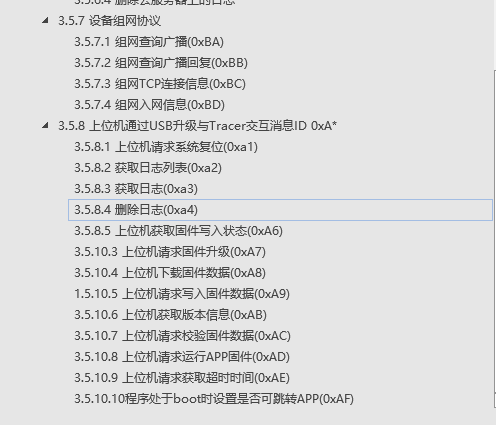
### class SysBase

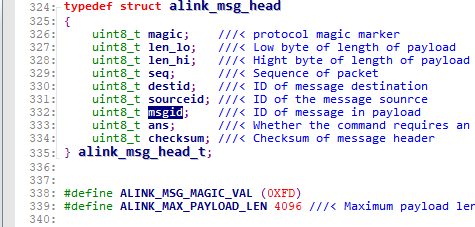
# alink消息转发

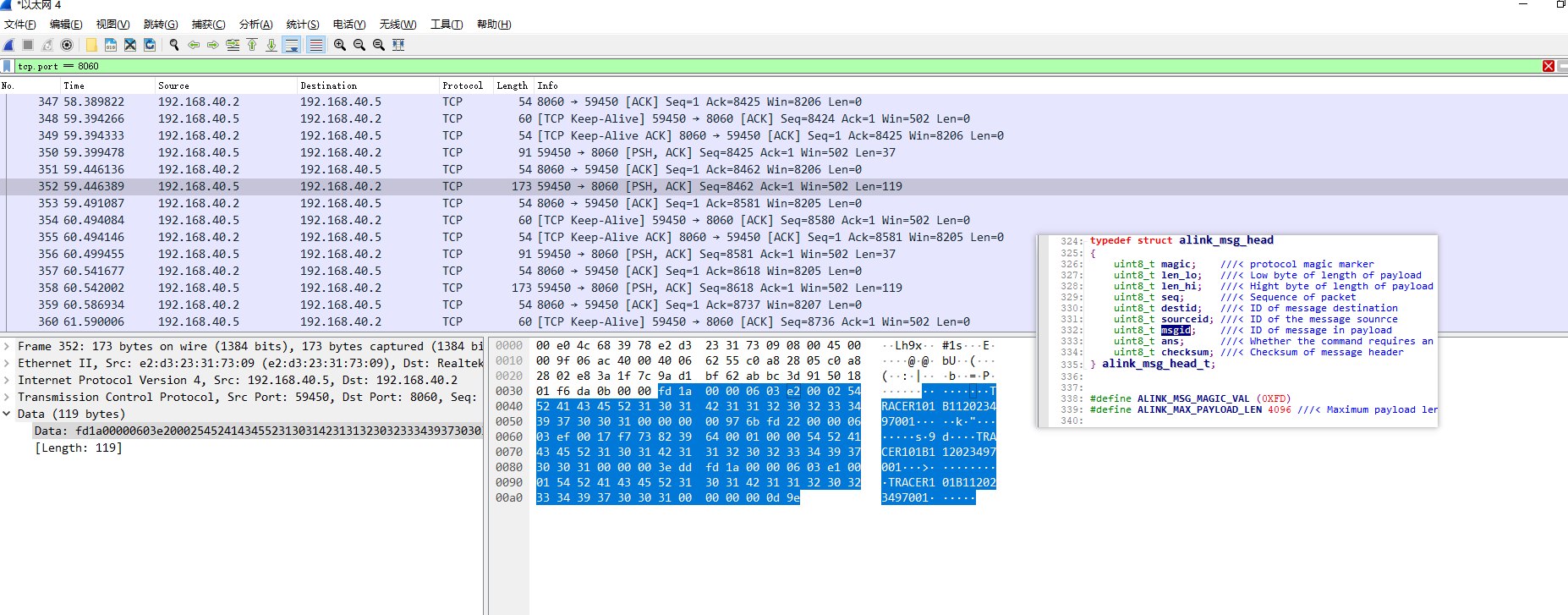


# Alink协议抓包





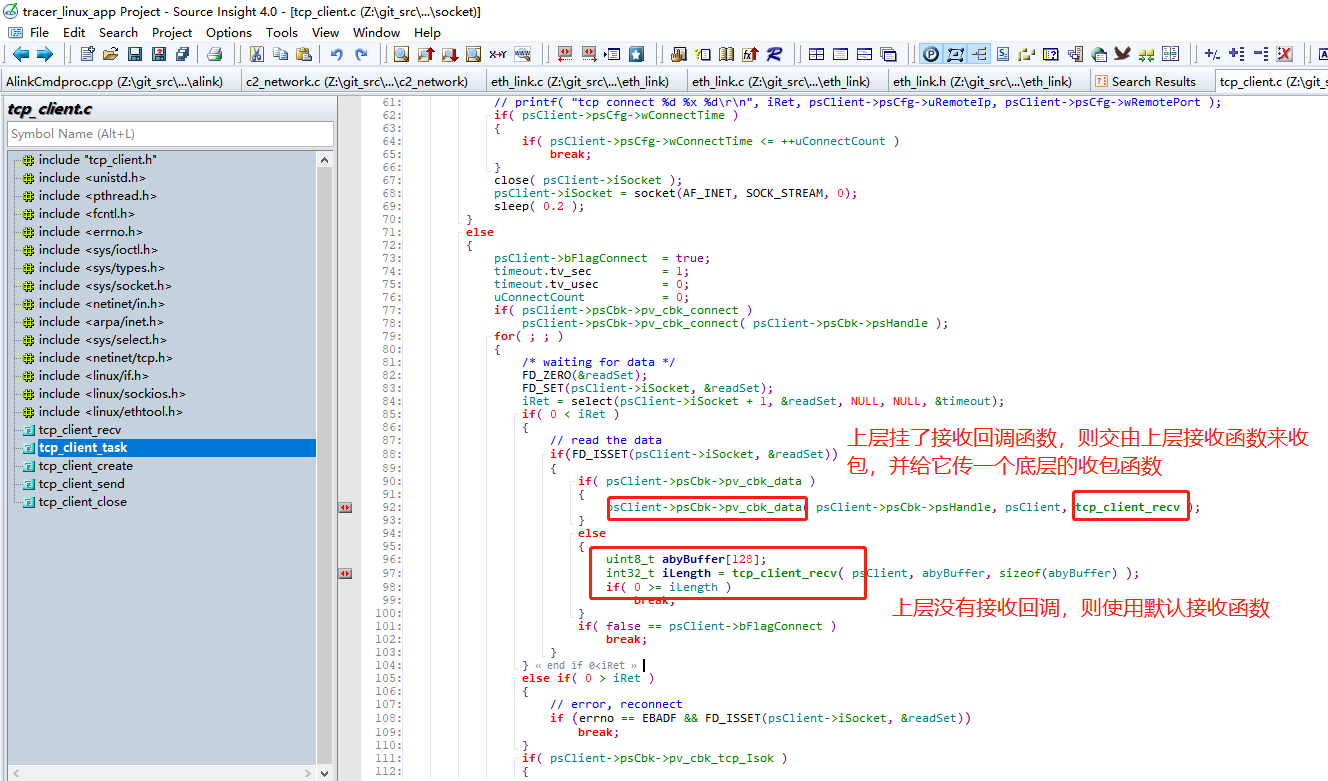




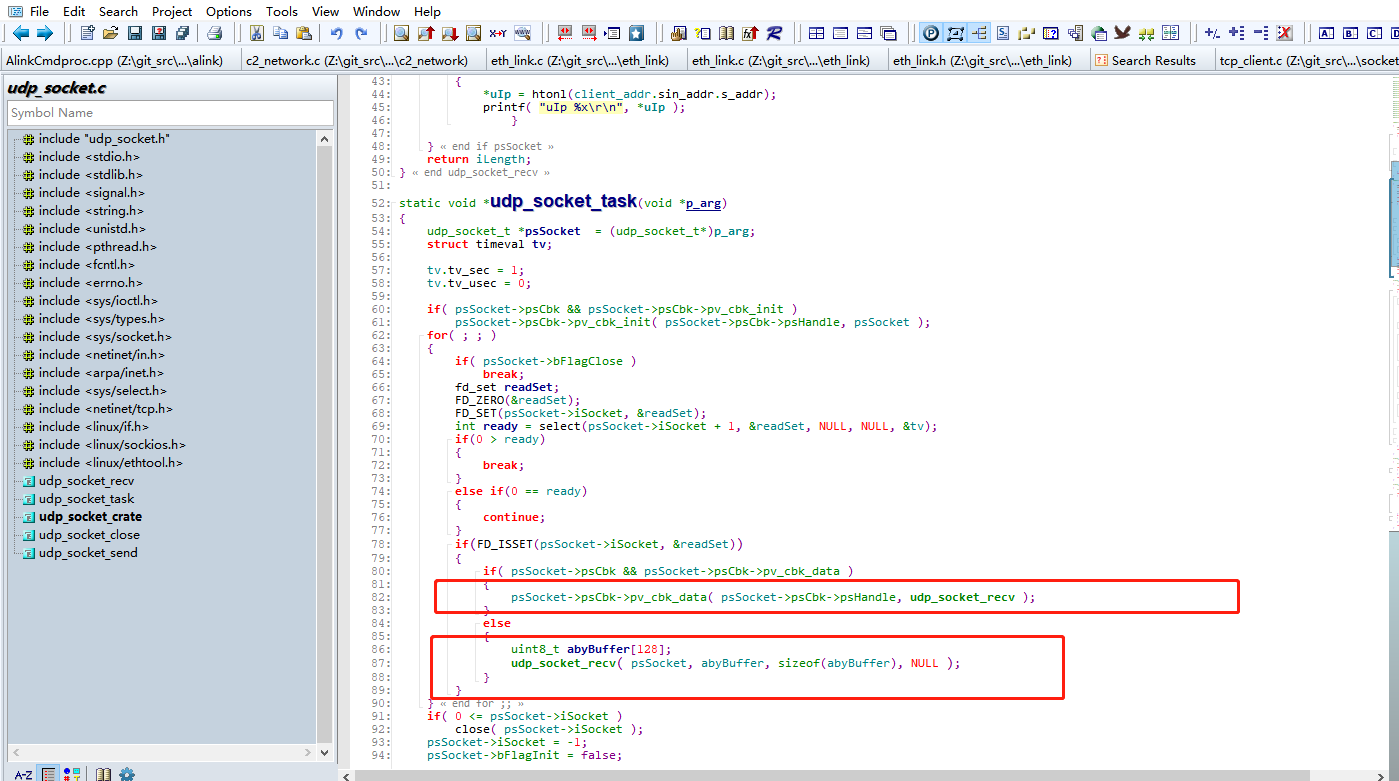
# Alink收发

## 底层接收数据（物理通讯链路层）

### tcp\_client.c

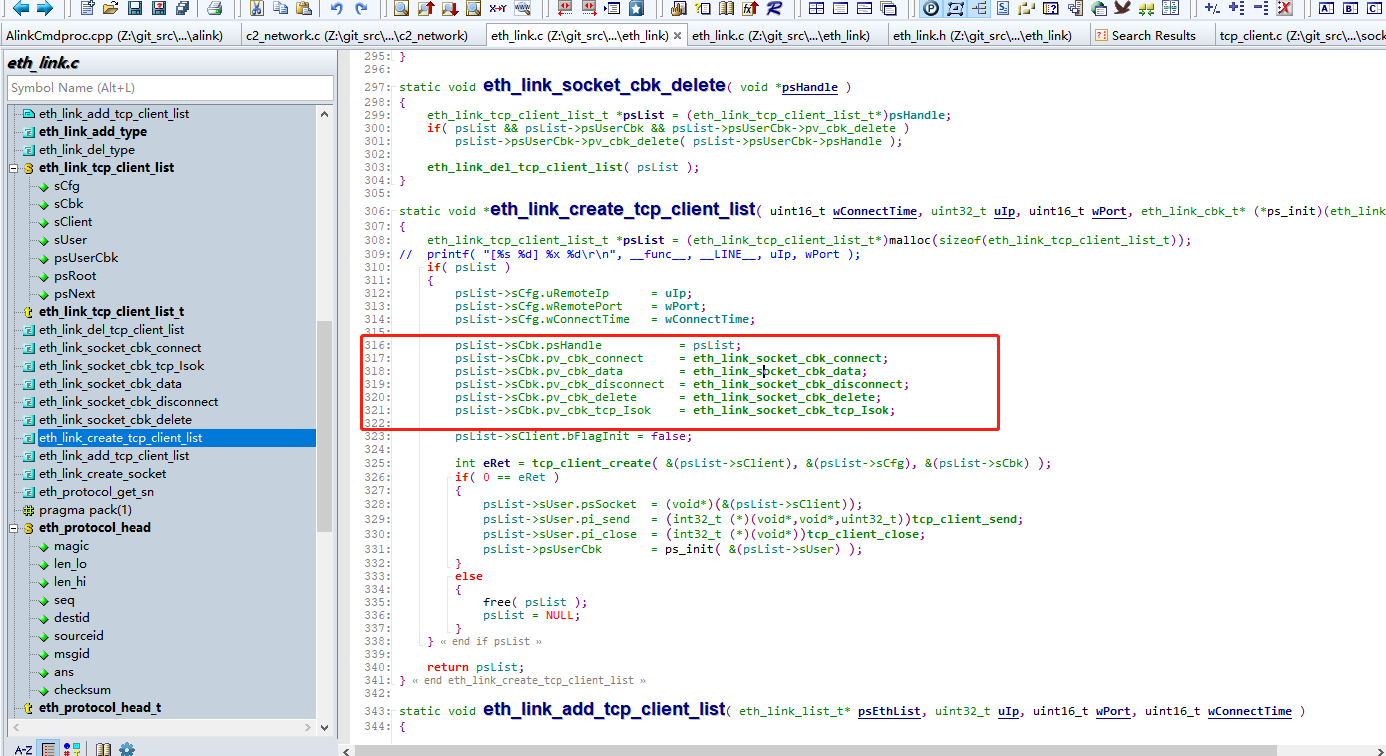


### udp\_socket.c

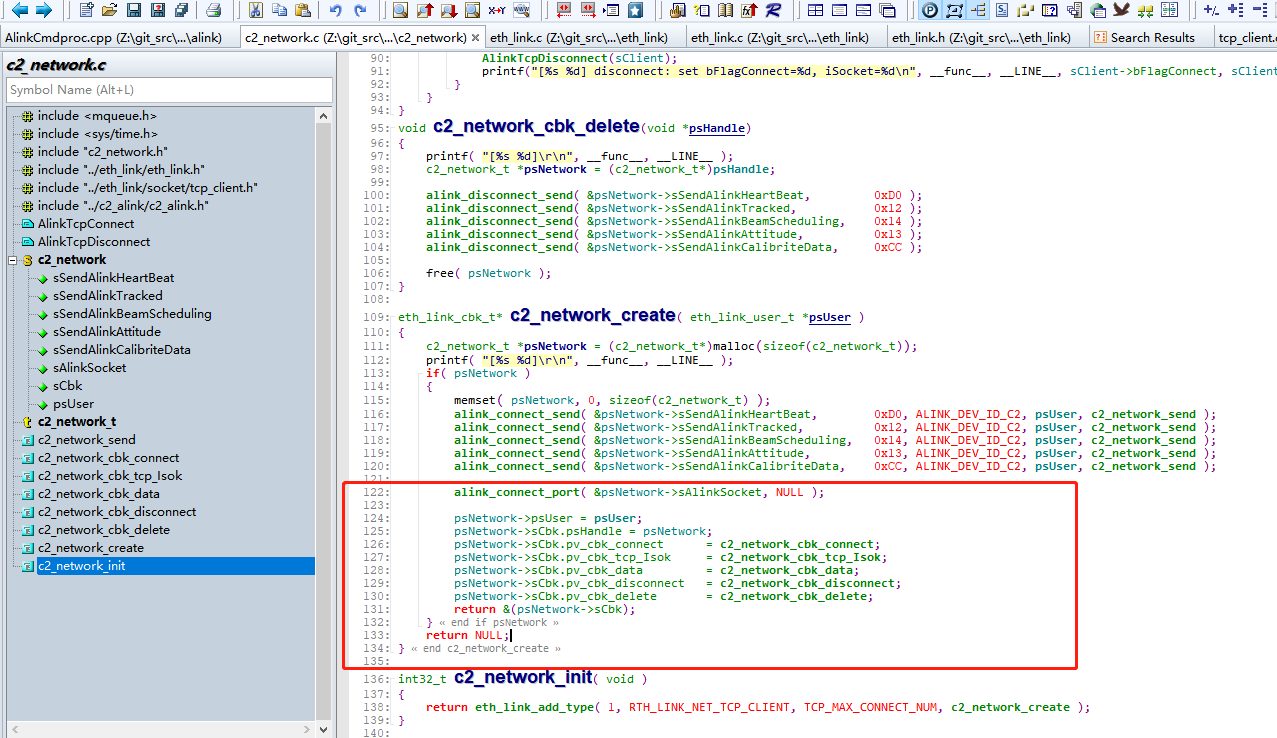


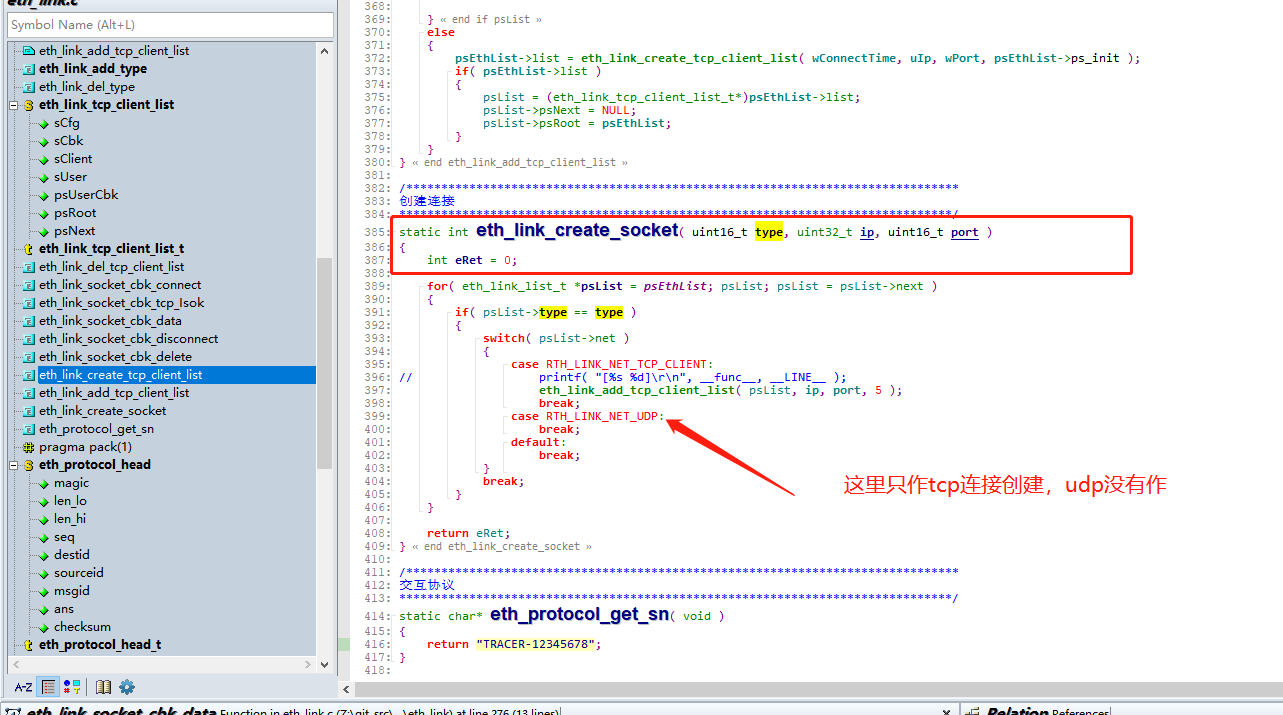
## 逻辑链路抽象层

### eth\_link.c

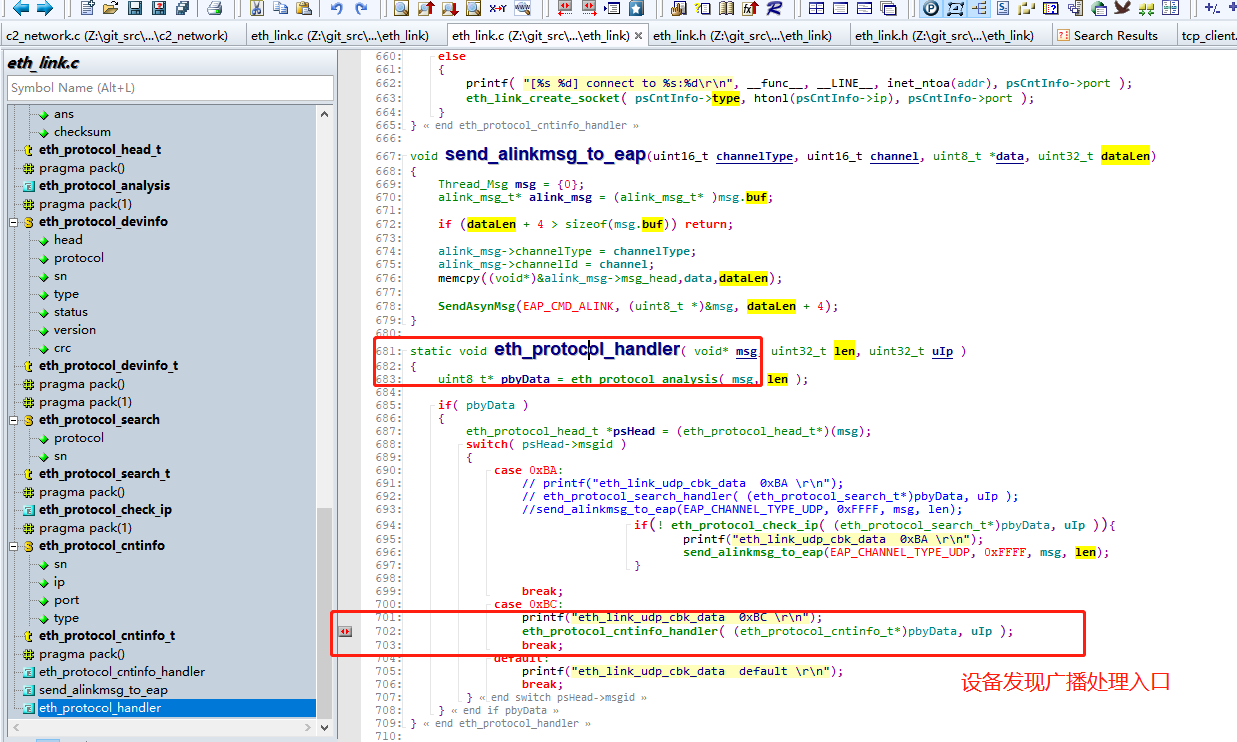


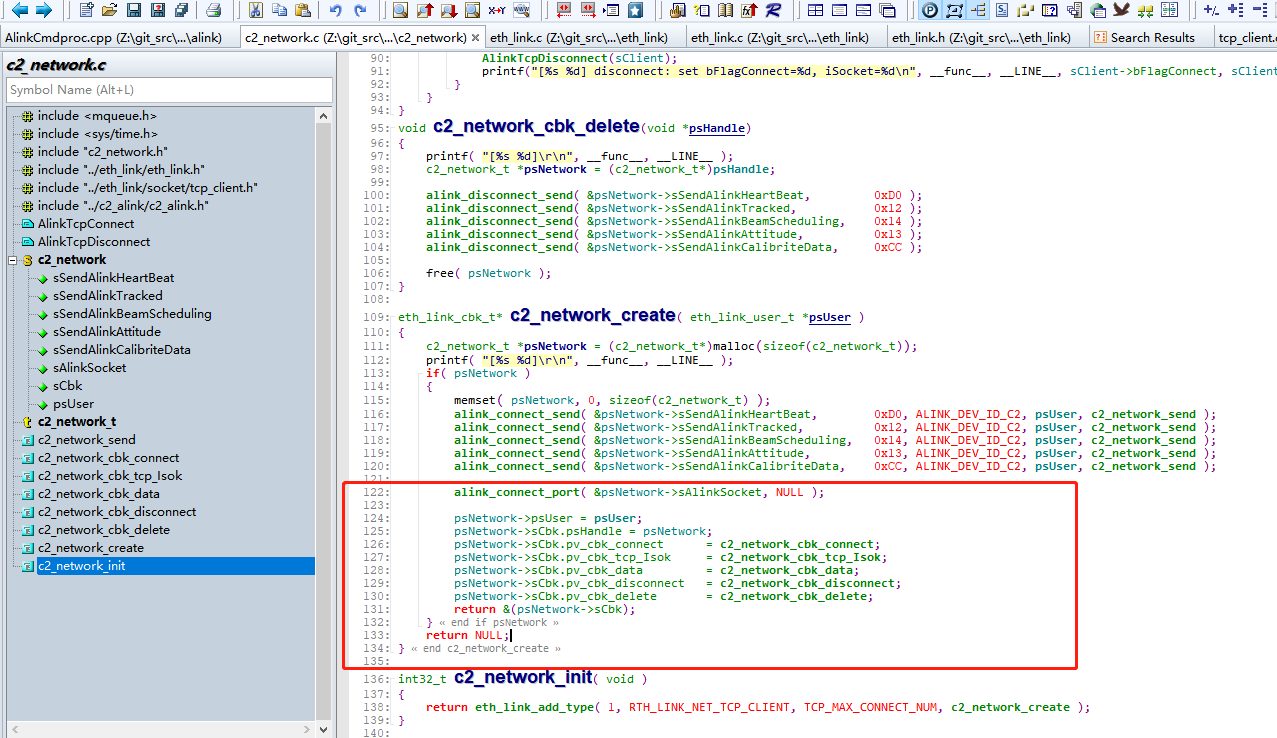
### c2\_network.c





## 设备发现广播协议的处理





eth\_link\_udp\_cbk\_data

eth\_protocol\_handler

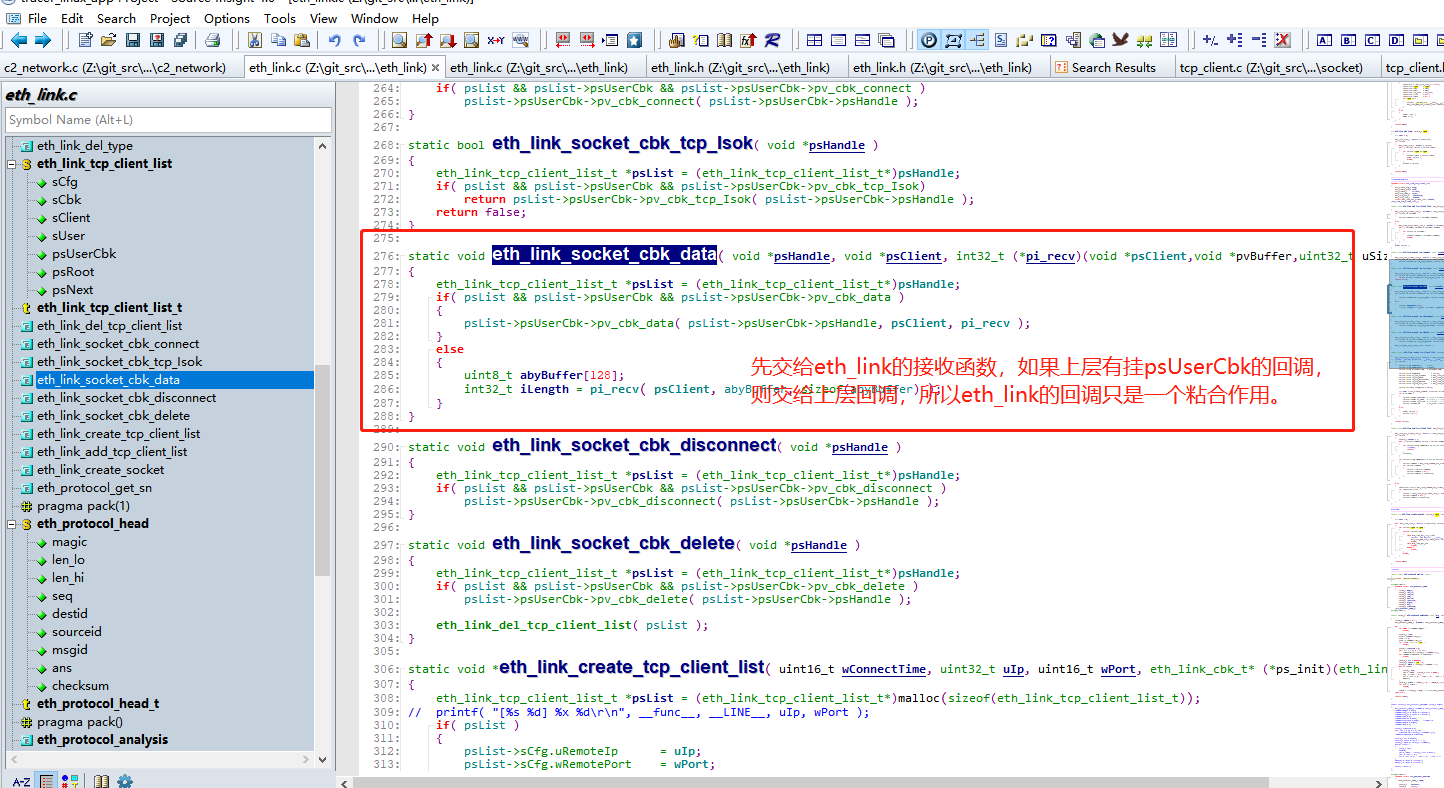
eth\_protocol\_cntinfo\_handler

eth\_link\_create\_socket

eth\_link\_add\_tcp\_client\_list (目前udp在这里处理，而是启动时就创建一个udp侦听口)

eth\_link\_create\_tcp\_client\_list

udp用于接收设备发现，发现后，创建tcp连接，可支持多路连接。



Tcp链路控制块链表结构：

typedef struct eth\_link\_tcp\_client\_list

{

tcp\_client\_cfg\_t sCfg;

tcp\_client\_cbk\_t sCbk;

tcp\_client\_t sClient;

eth\_link\_user\_t sUser;

eth\_link\_cbk\_t \*psUserCbk;

eth\_link\_list\_t \*psRoot;

struct eth\_link\_tcp\_client\_list \*psNext;

}eth\_link\_tcp\_client\_list\_t;

Alink相关控制结构

typedef struct c2\_network

{

alink\_send\_list\_t sSendAlinkHeartBeat;

alink\_send\_list\_t sSendAlinkTracked;

alink\_send\_list\_t sSendAlinkBeamScheduling;

alink\_send\_list\_t sSendAlinkAttitude;

alink\_send\_list\_t sSendAlinkCalibriteData;

alink\_socket\_t sAlinkSocket;

eth\_link\_cbk\_t sCbk;

eth\_link\_user\_t \*psUser;

}c2\_network\_t;

typedef struct eth\_link\_cbk

{

void \*psHandle;

void (\*pv\_cbk\_connect)(void \*psHandle);

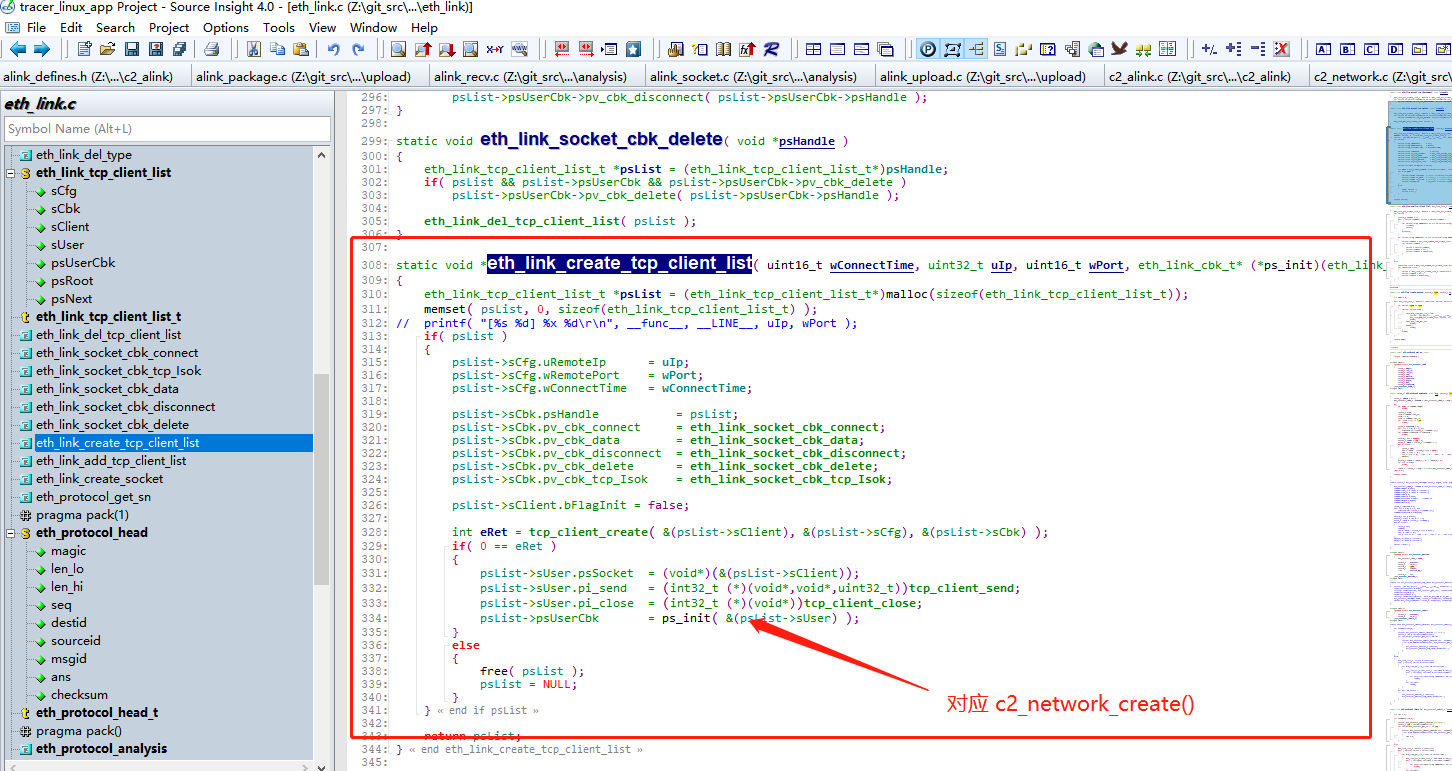
void (\*pv\_cbk\_data)(void \*psHandle,void \*psClient,int32\_t (\*pi\_recv)(void \*psClient,void \*pvBuffer,uint32\_t uSize));

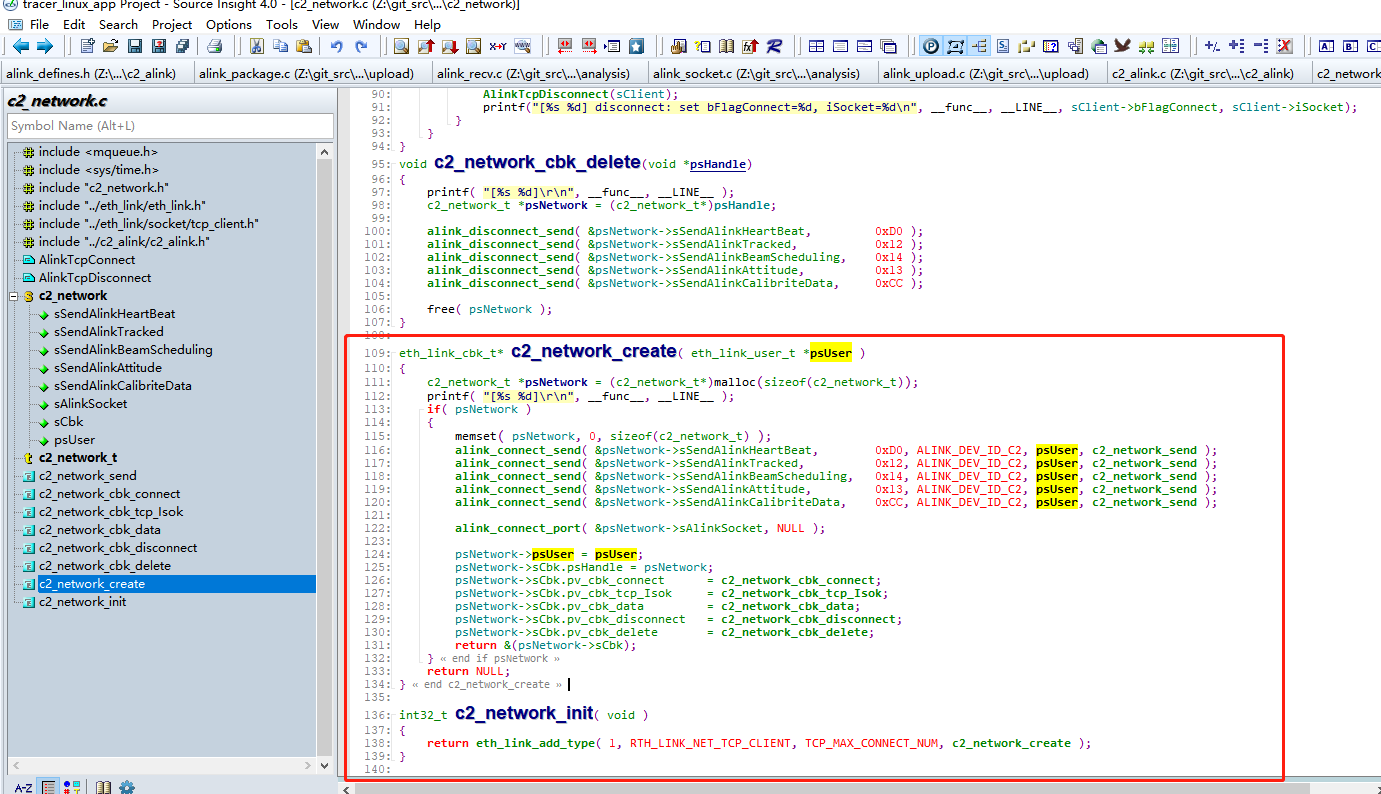
void (\*pv\_cbk\_disconnect)(void \*psHandle);

void (\*pv\_cbk\_delete)(void \*psHandle);

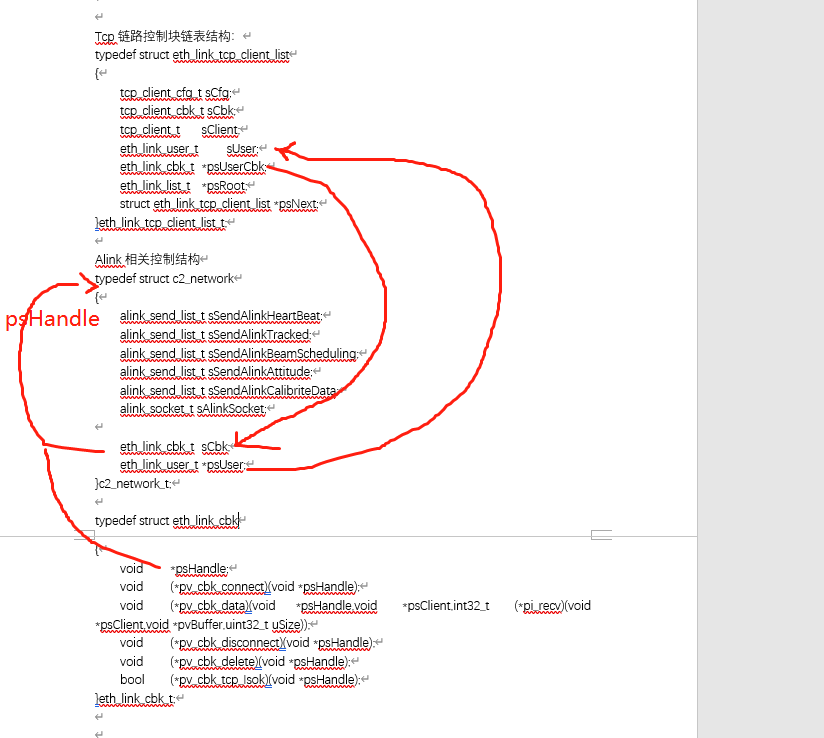
bool (\*pv\_cbk\_tcp\_Isok)(void \*psHandle);

}eth\_link\_cbk\_t;





这两个结构如何关联呢：



接收过程：

# 升级

## 升级相关的重要文件

modified: ../../linux\_app/src/eap/alink/AlinkCmdproc.cpp

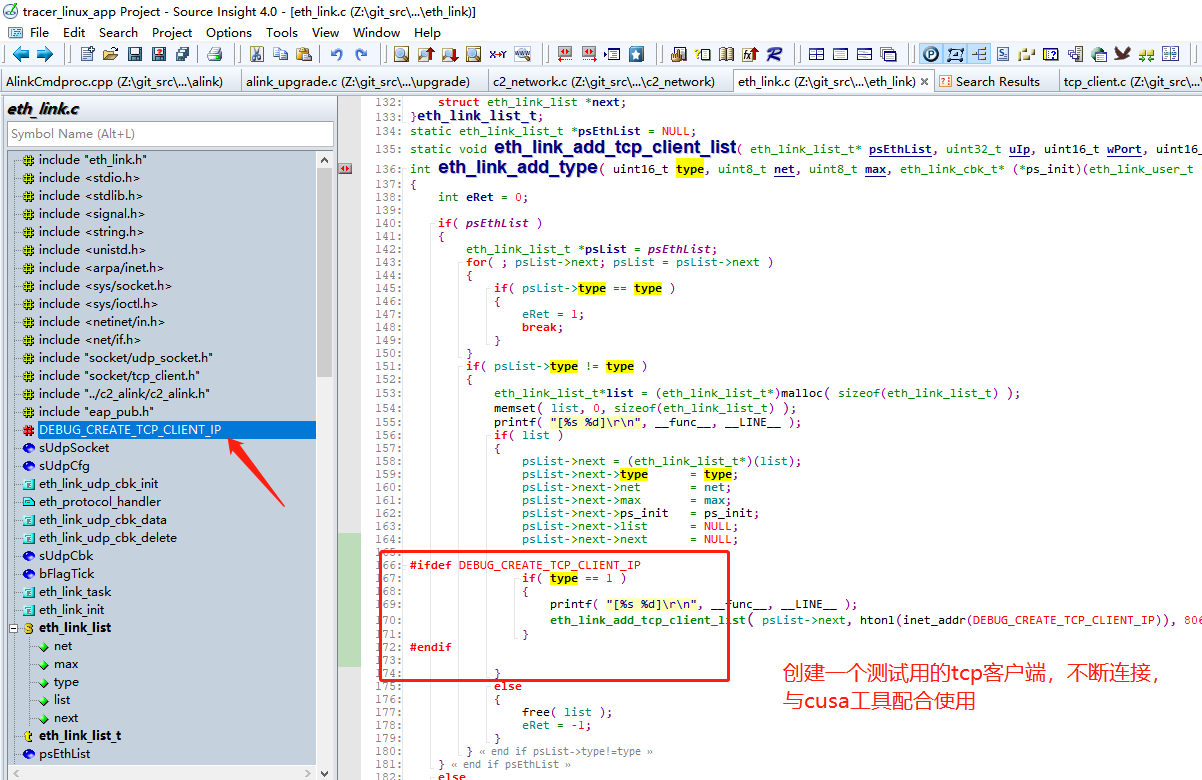
modified: ../../linux\_app/src/tracer/TracerCommonHal.cpp

modified: ../../linux\_app/src/tracer/hal/c2\_alink/upgrade/alink\_upgrade.c

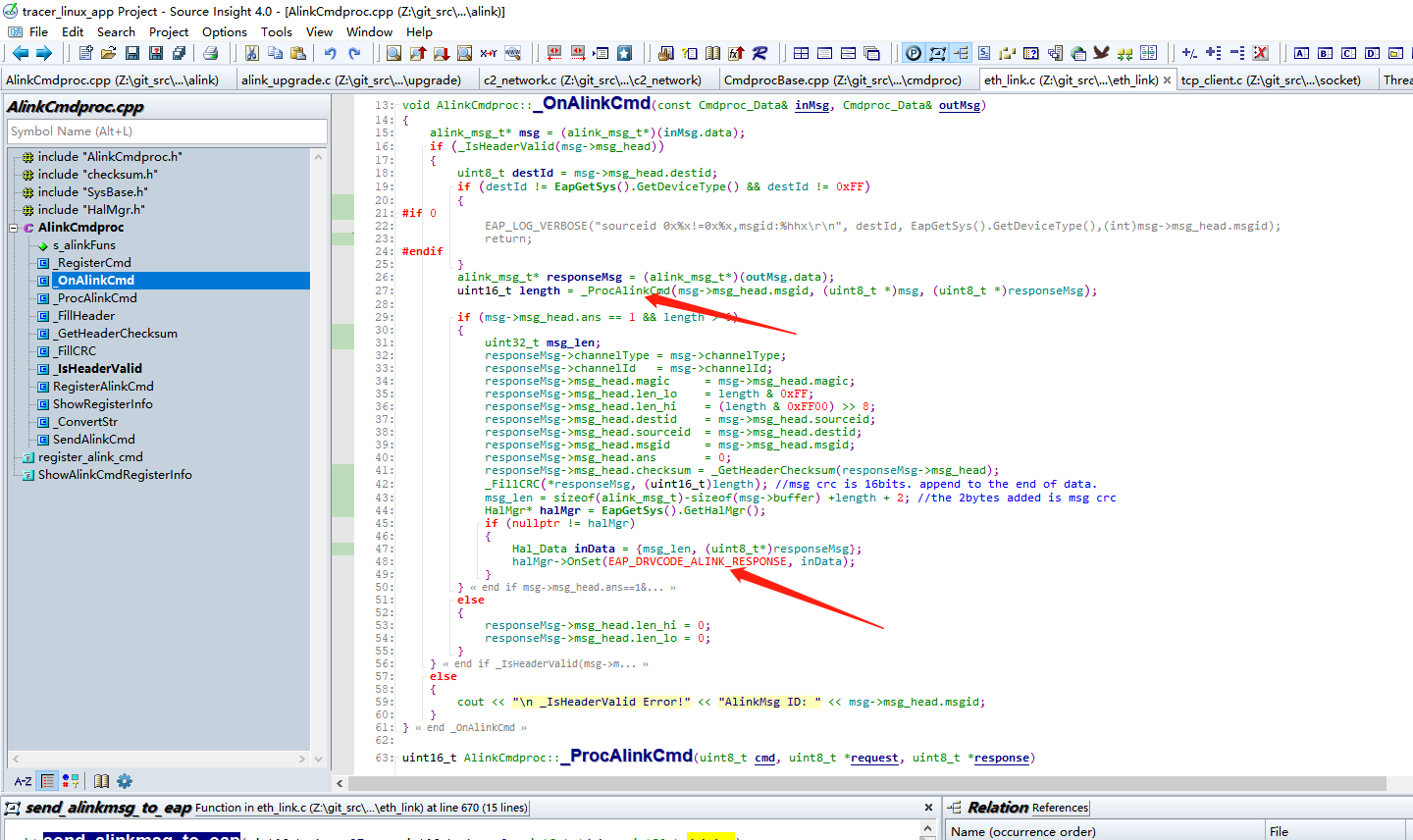
modified: ../../linux\_app/src/tracer/hal/c2\_network/c2\_network.c

modified: ../../linux\_app/src/tracer/hal/eth\_link/eth\_link.c

modified: ../../linux\_app/src/tracer/hal/eth\_link/socket/tcp\_client.c



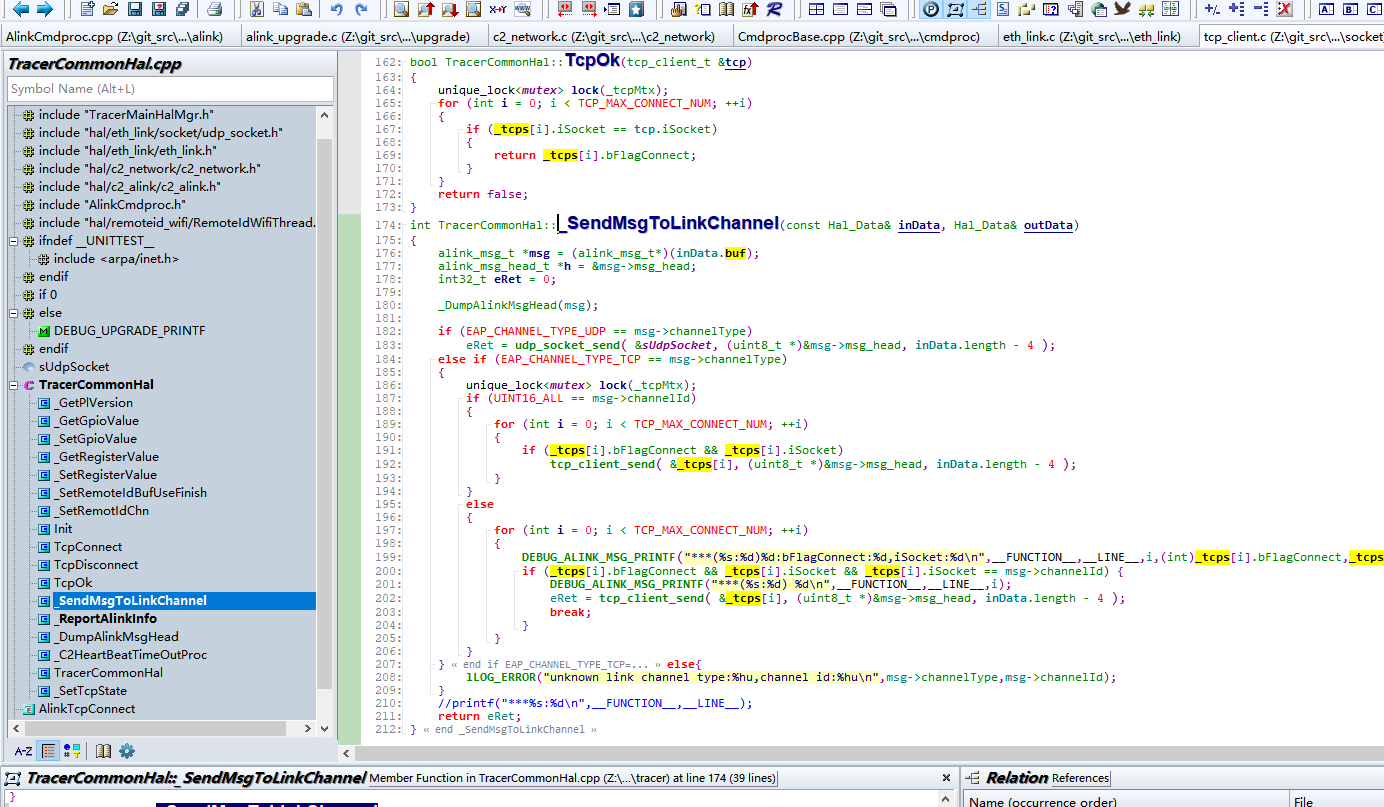
### AlinkCmdproc.cpp



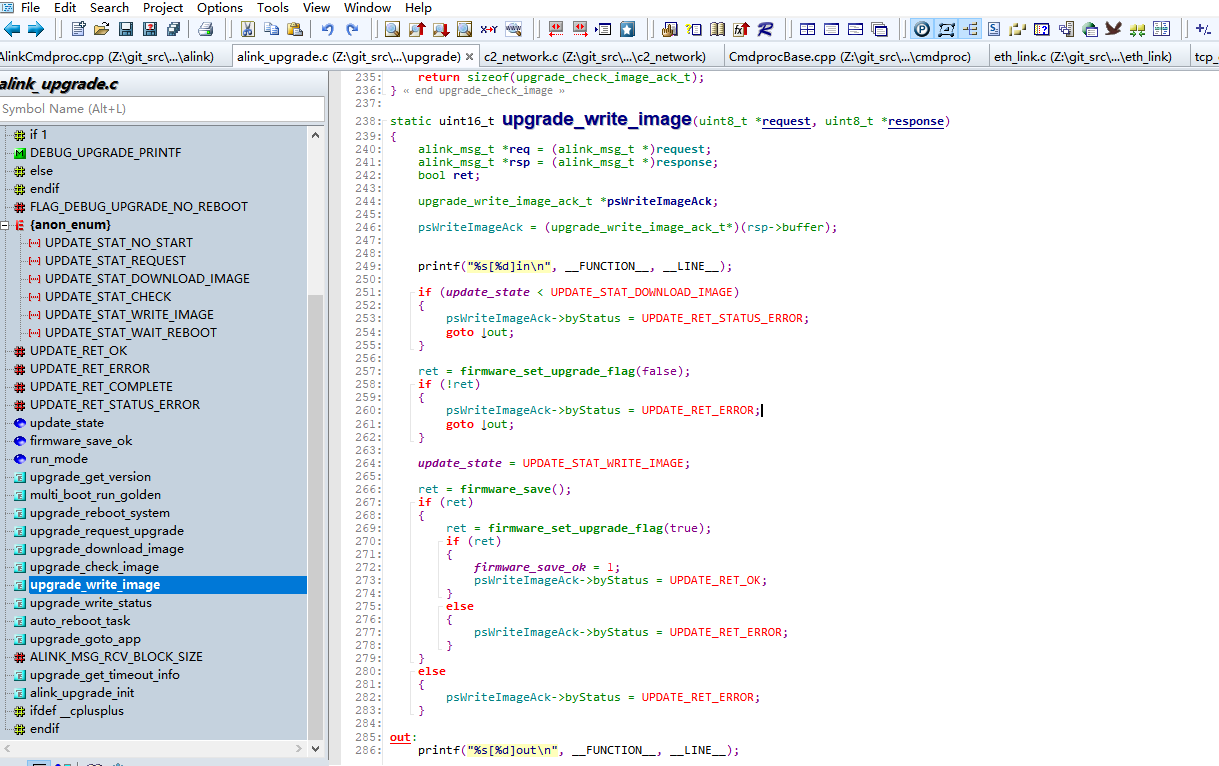
Alink消息派发入口，以及应答消息处理。

应答消息，要注意链路信息。保证从哪个链路进入，从哪个链路发出。

### TracerCommonHal.cpp

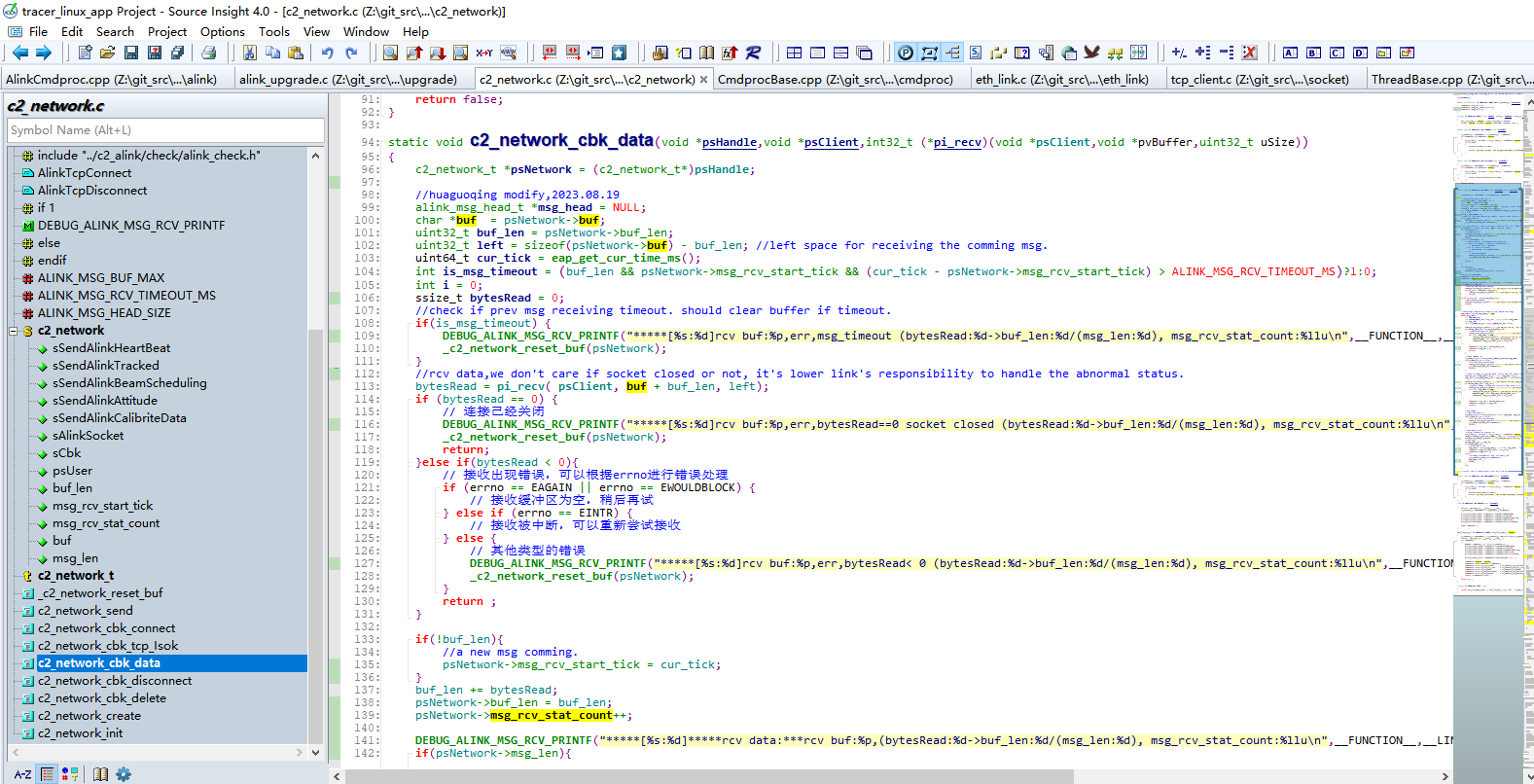


### alink\_upgrade.c



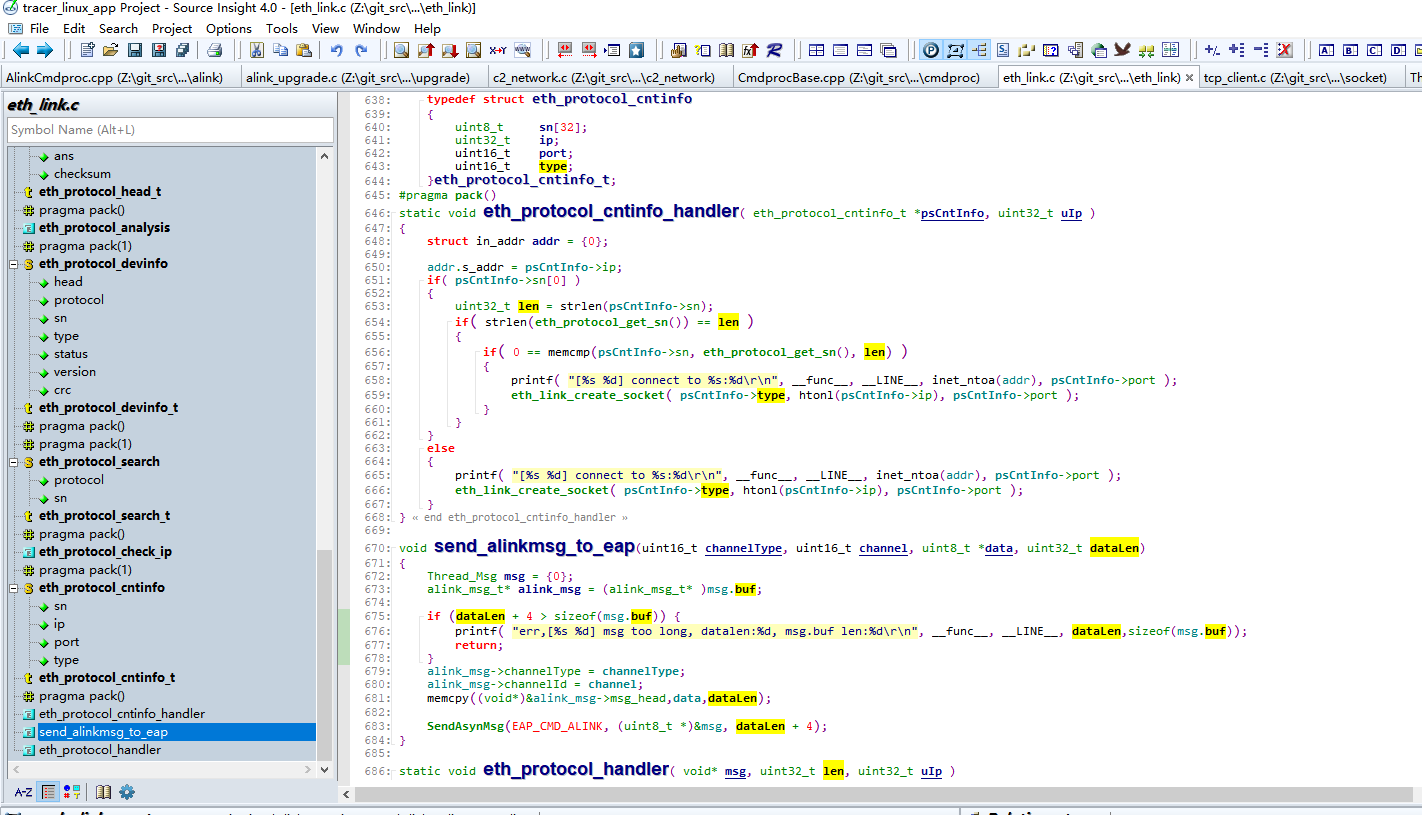
升级alink相关的消息处理

### c2\_network.c



Alink消息拆包，关键是处理好粘包问题。

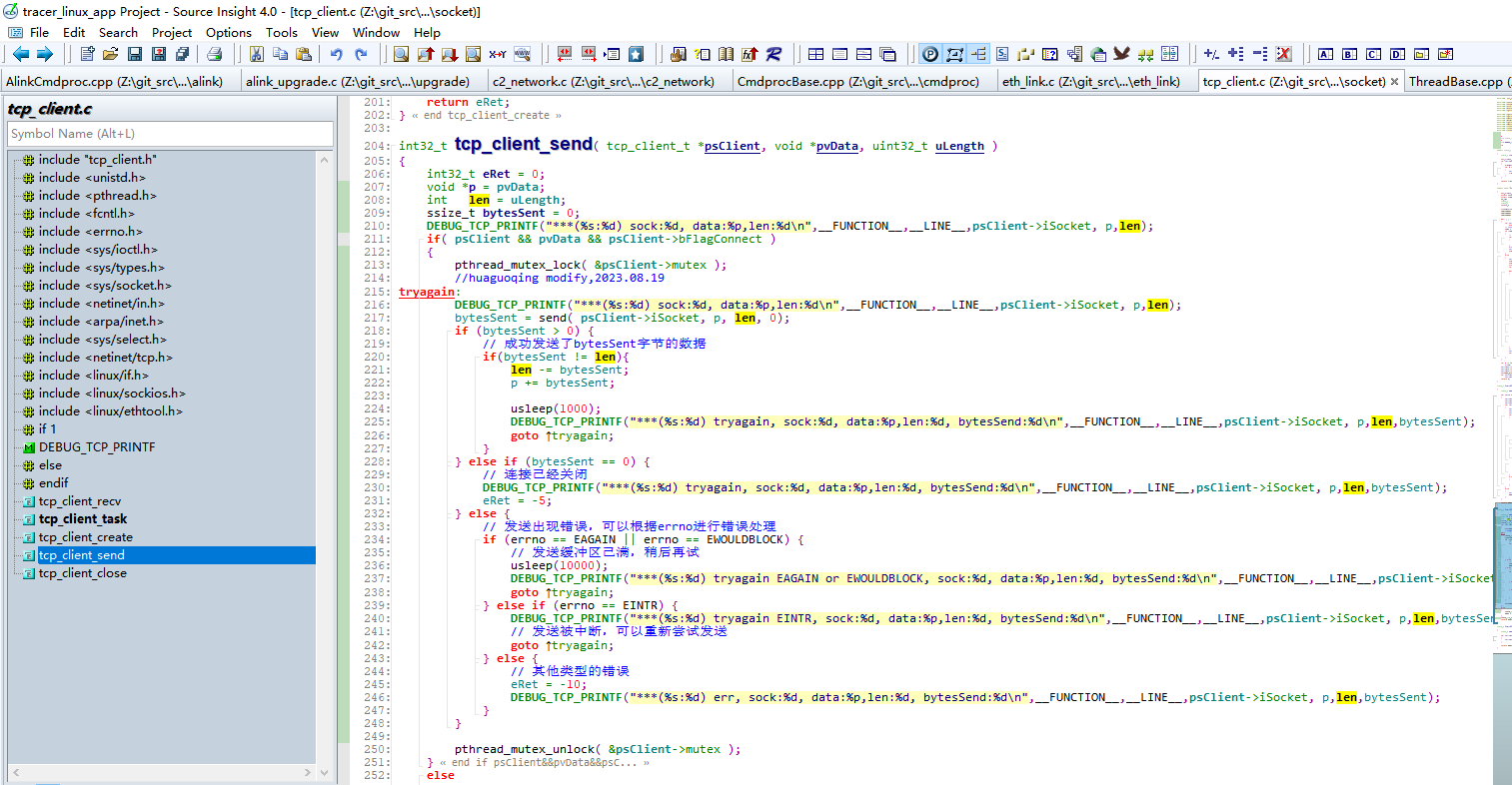
### eth\_link.c



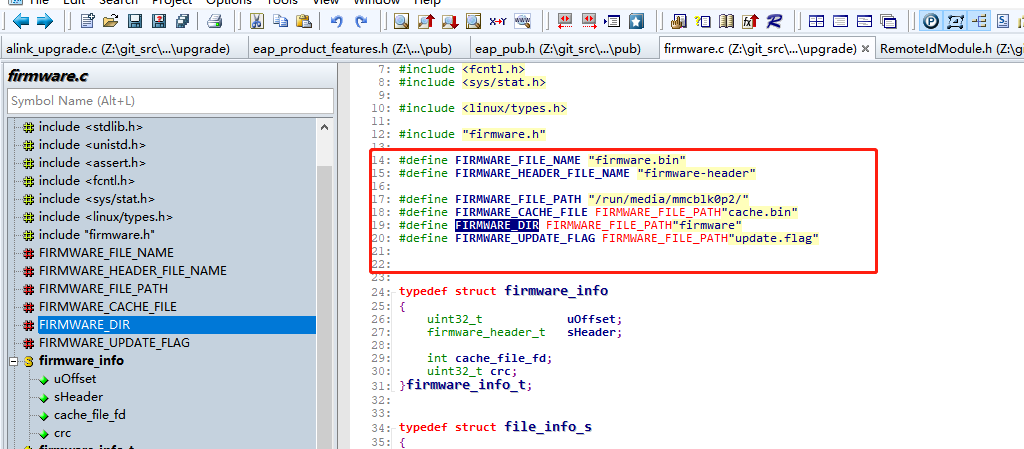
Udp设备发现的处理

发现设备后，创建tcp链接，对多路链接进行管理

### tcp\_client.c



### 升级文件存储

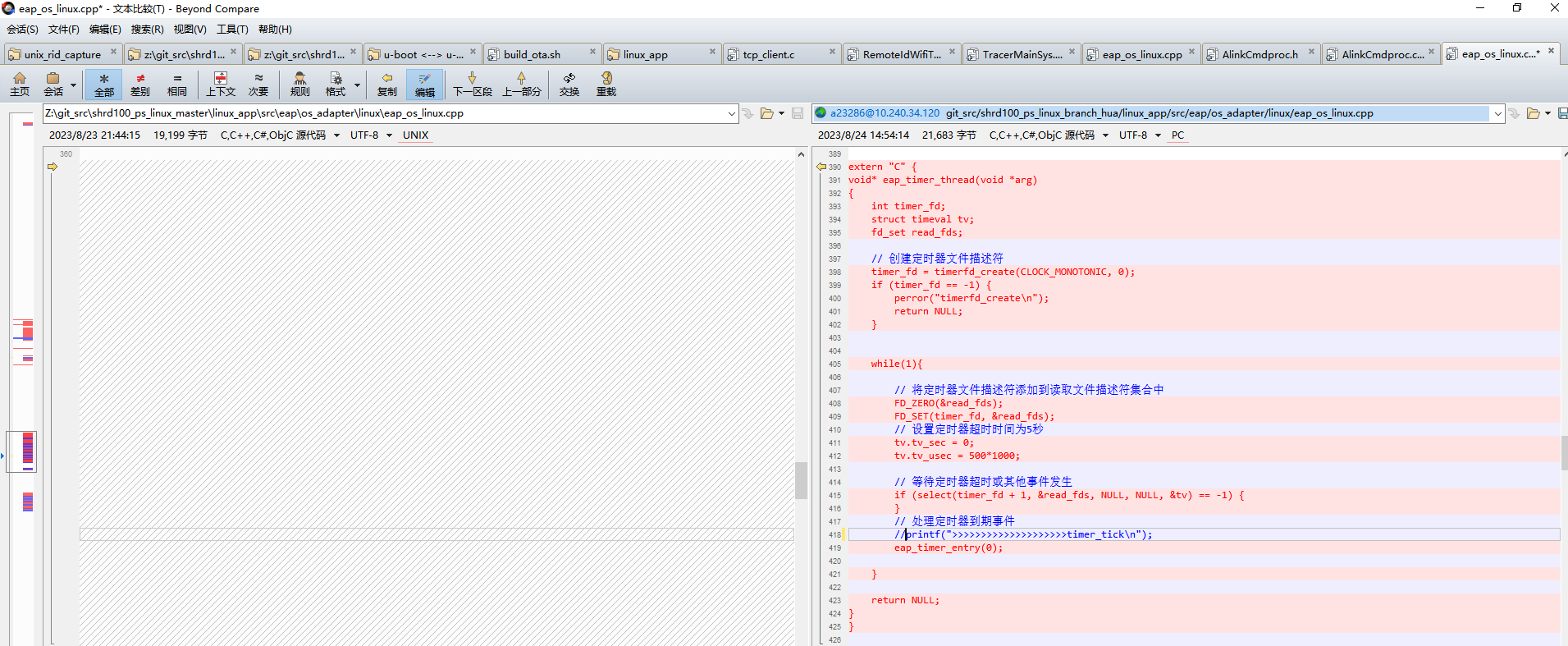


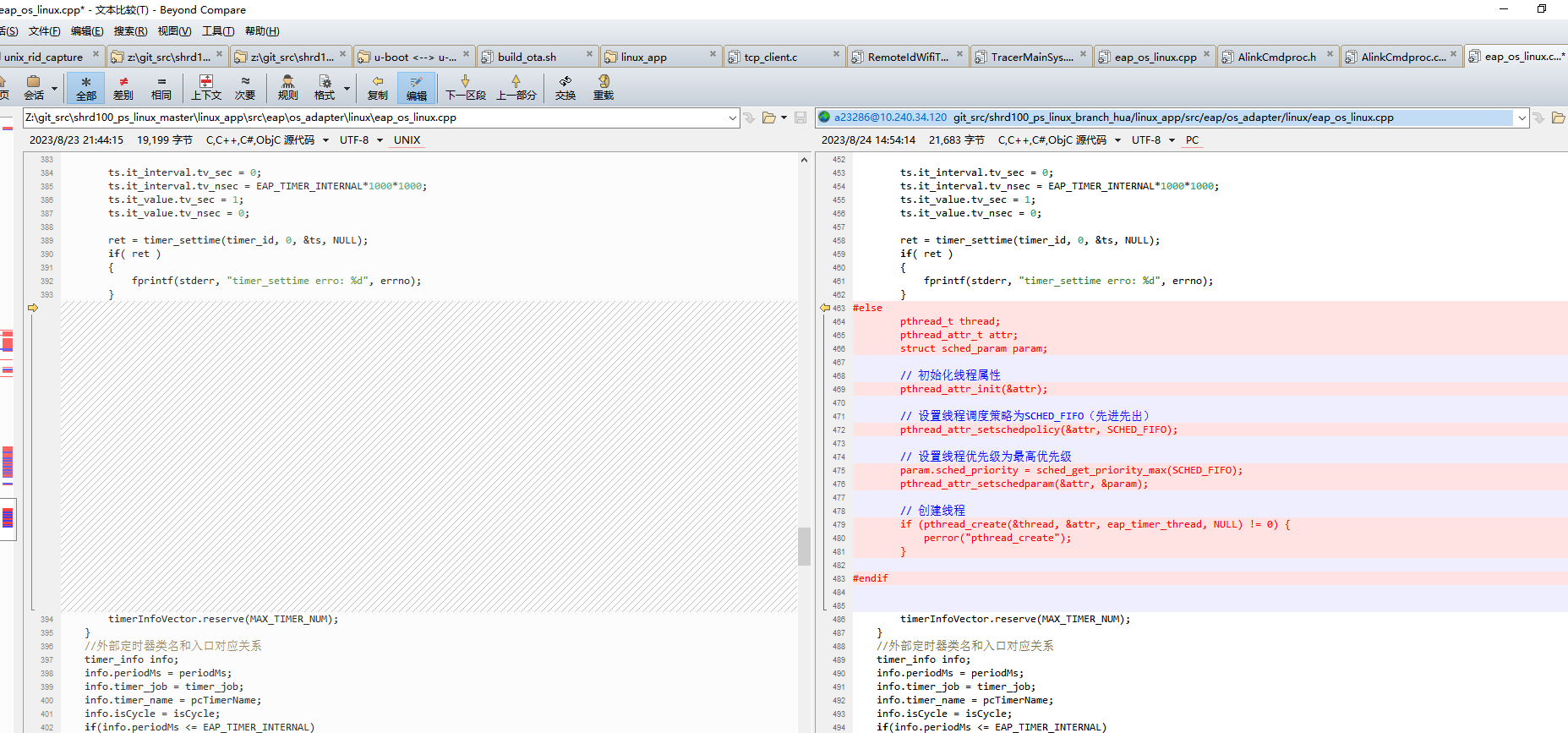
下载下来后，暂存在mmcblk0p2中 cache.bin

解开文件放在fireware目录里。

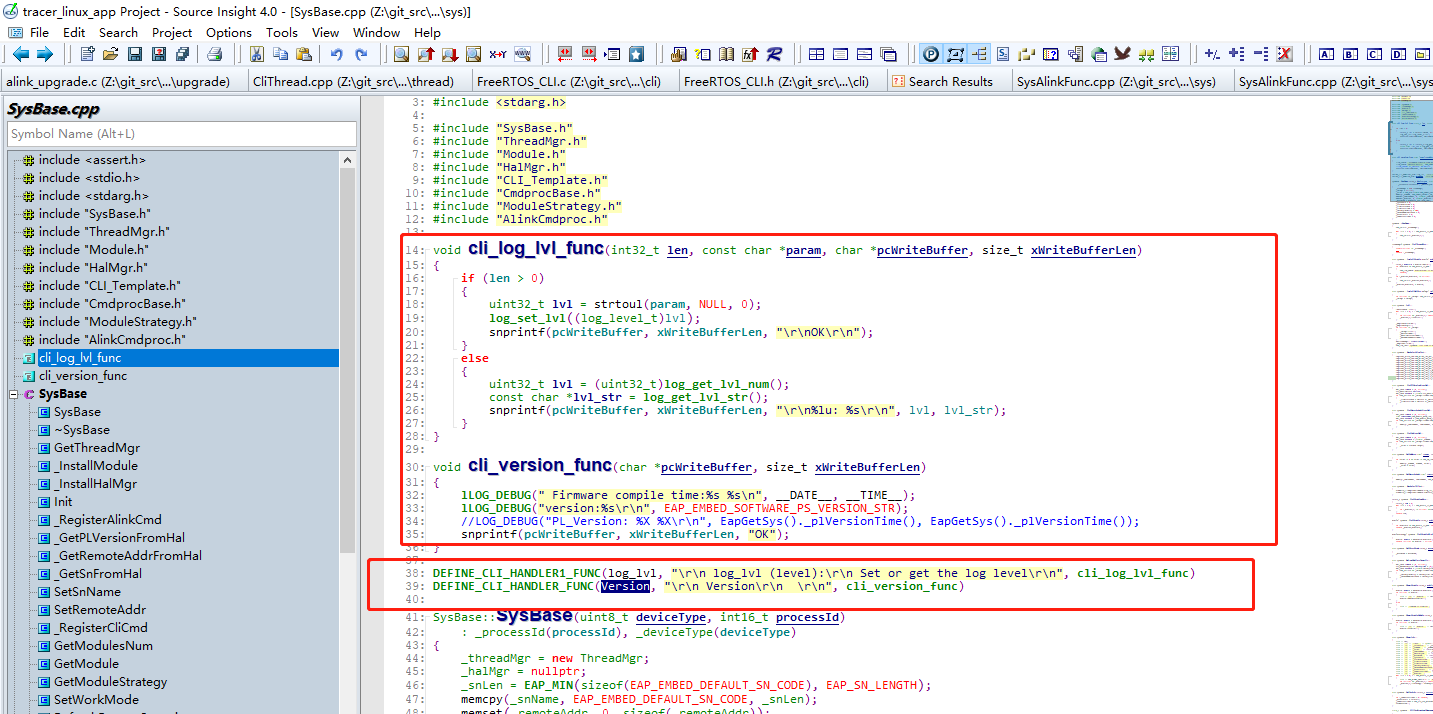
升级标记文件是update.flag. 内容是”upgrade”字符串

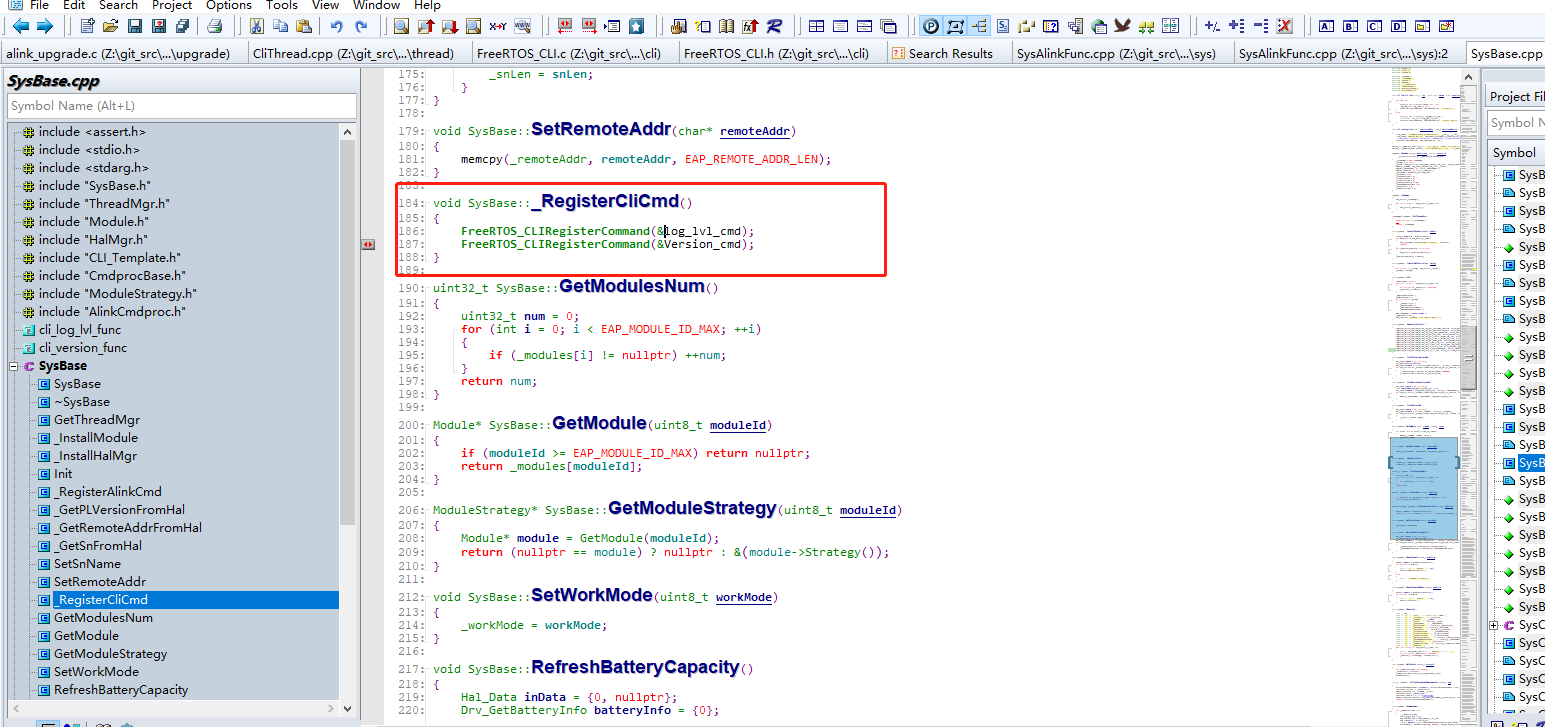
# 定时器





# Cli命令处理





FreeRTOS\_CLIRegisterCommand()

## Wificfg cli命令

ShowCliRegisteredCmds()

wificfg.c

注意0x2f alink消息处理函数

Alink\_C2CliCmd\_Func()

## telnet cli

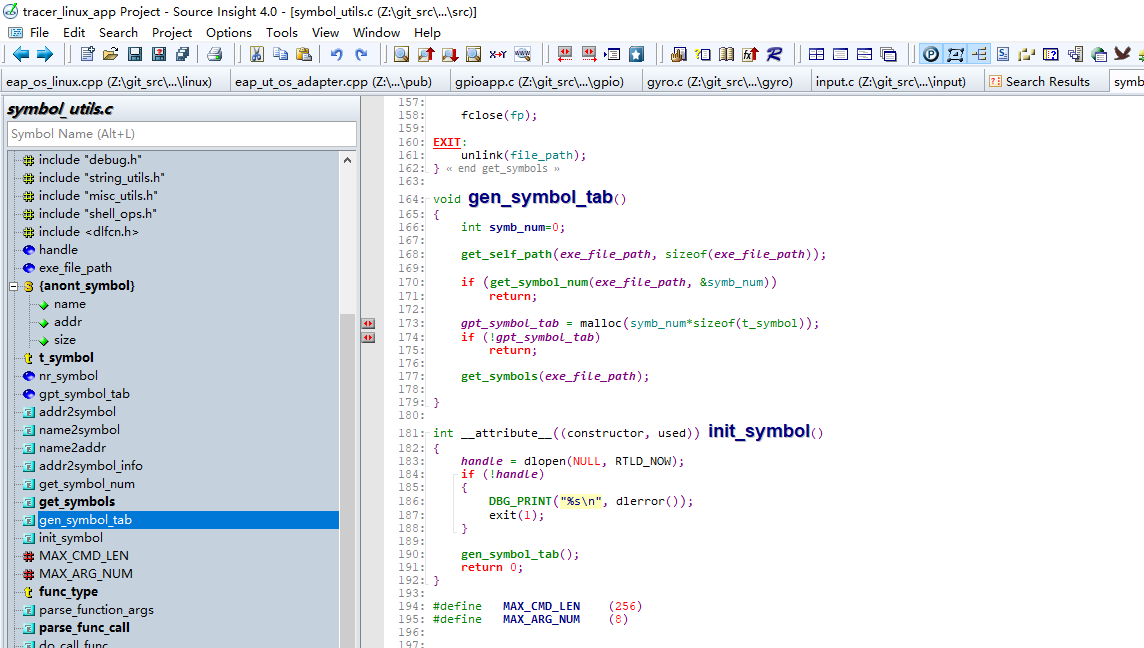
[GitHub - sunmingbao/linux-dev-framework: A framework for quick start app develop work using c language with gcc under linux](https://github.com/sunmingbao/linux-dev-framework/tree/master)

sunmingbao

### parse\_func\_call

symbol\_utils.c

name2addr



全局变量或函数的指针获取方法，采用读取/proc/self/exe,然后使用nm获取地址和函数名的对应关系。

# 链路心跳机制

当前实现的链路心跳机制是有问题的，应该每个链路要有自己的心跳。而不是typeC一个，tcp一个。因为有可以同时存在多个tcp链接，所以心跳的监测数据，应该放在链路层维护。

# telnet4debug功能使用

telnet localhost 10000

# 模块通讯机制

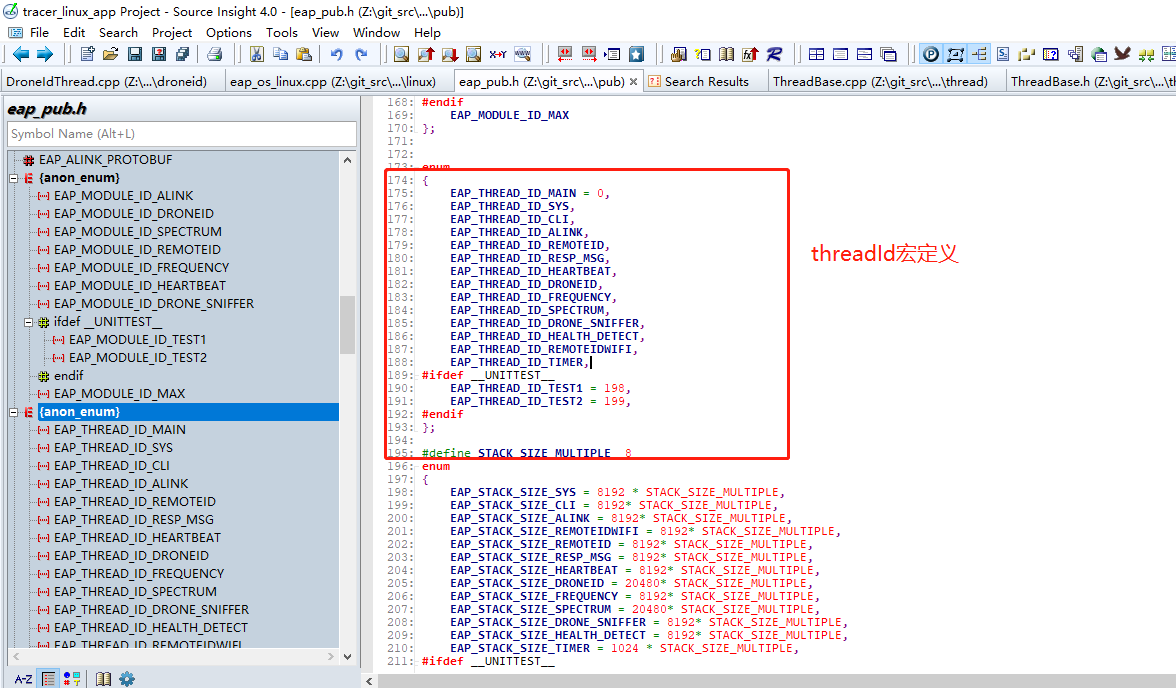
ipcs -q | cut -d ' ' -f 2 | xargs -I {} ipcrm -q {}

### Threadid的定义

并不真正的pthread的thread id;而是我们内部的宏定义编号。

滑国青：这个threadid是全设备唯一，还是进程内唯一呢？

如果是全设备唯一，那是不是说就无法在不同的进程内创建同一种threadid的module了呢？



Thread index与thread id的关系：

所有的module都有自己的thread id;但程序启动时，并不需要所有module运行。

### Remote的概念

Remote就是queue handle; 是mq\_open创建队列返回的队列句柄

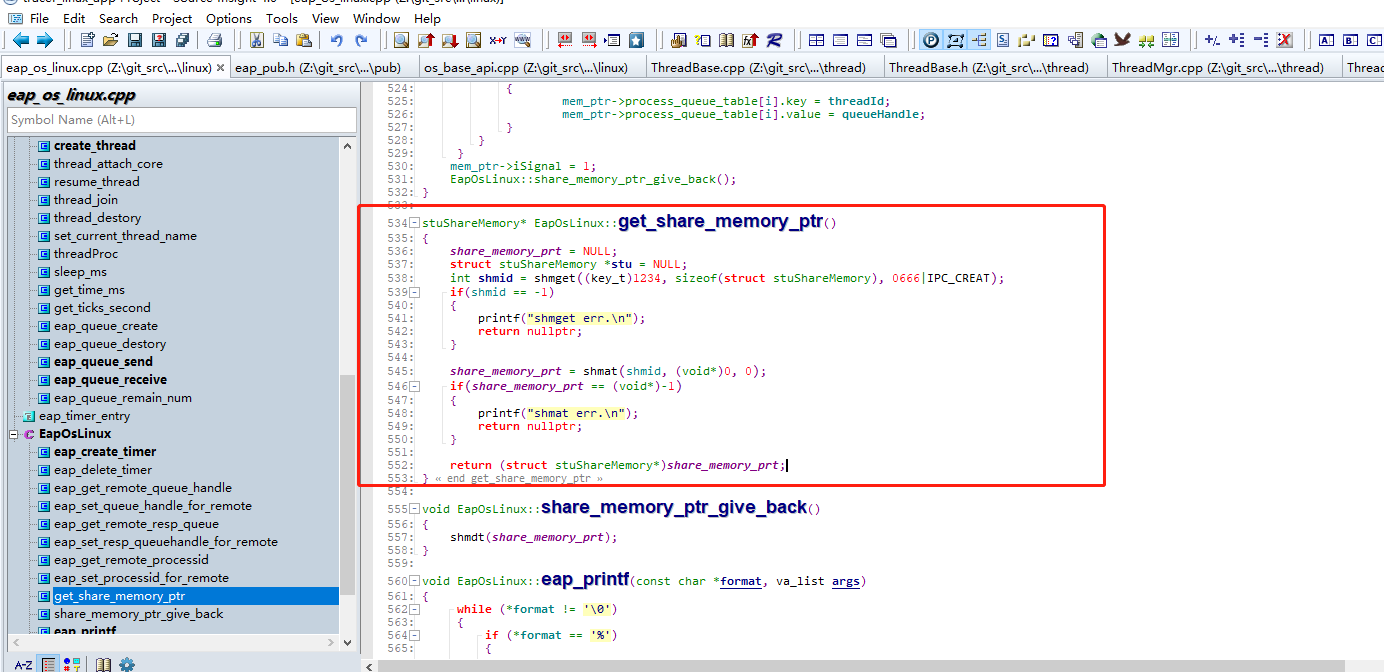
## 共享内存保存队列信息

### 共享内存指针获取

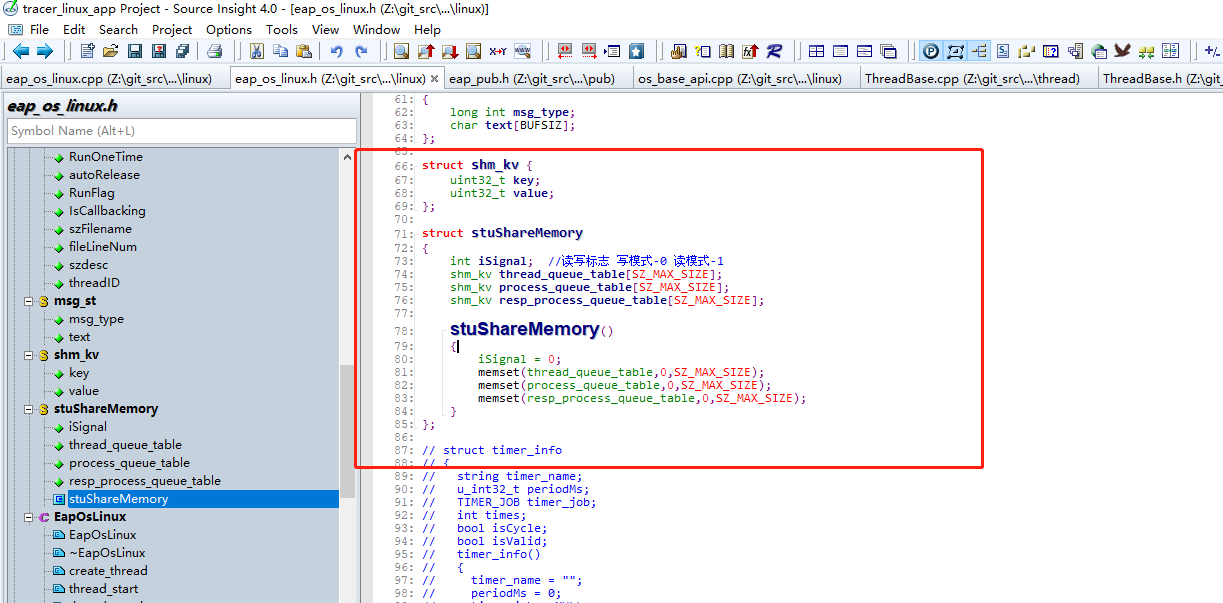
共享内存的key为1234,即4d2

通过ipcs可以看到





### 共享内存数据结构



Thread\_queue\_table:

Key是threadId, value是queue handle

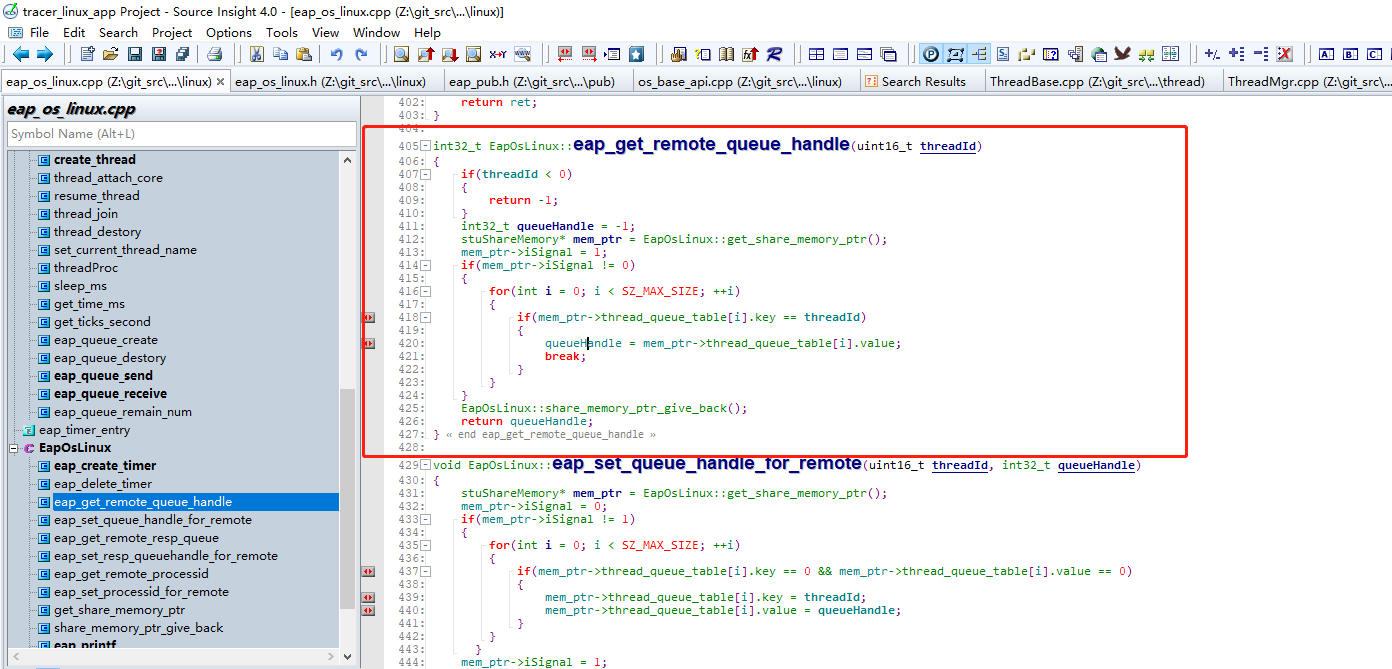
所以是线程id与queue id的映射表。

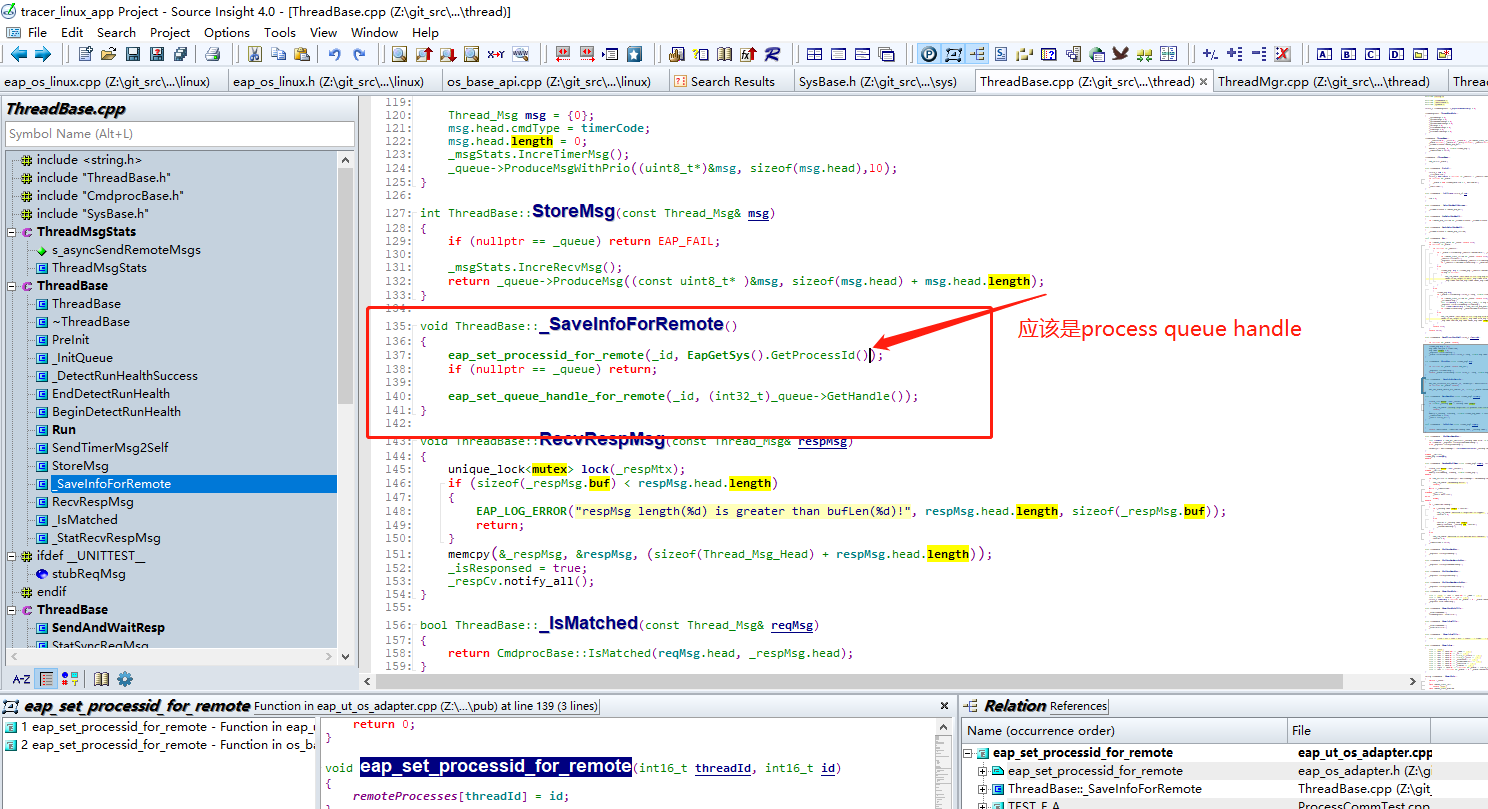
滑国青：这张表放在共享内存里，没有意义，因为在threadMgr里已经有了，而且没必要跨进程共享。 目前来看，除了process\_queue\_table,用于在多个进程间共享信息，其它两张表thread\_queue\_table和resp\_process\_queue\_table都没有用。

Process\_queue\_table:

Key: threadid

Value: process queue handle.





## eap\_os\_linux.cpp

eap\_queue\_create

mq\_open

eap\_queue\_destory

eap\_queue\_send

mq\_timedsend

eap\_queue\_receive

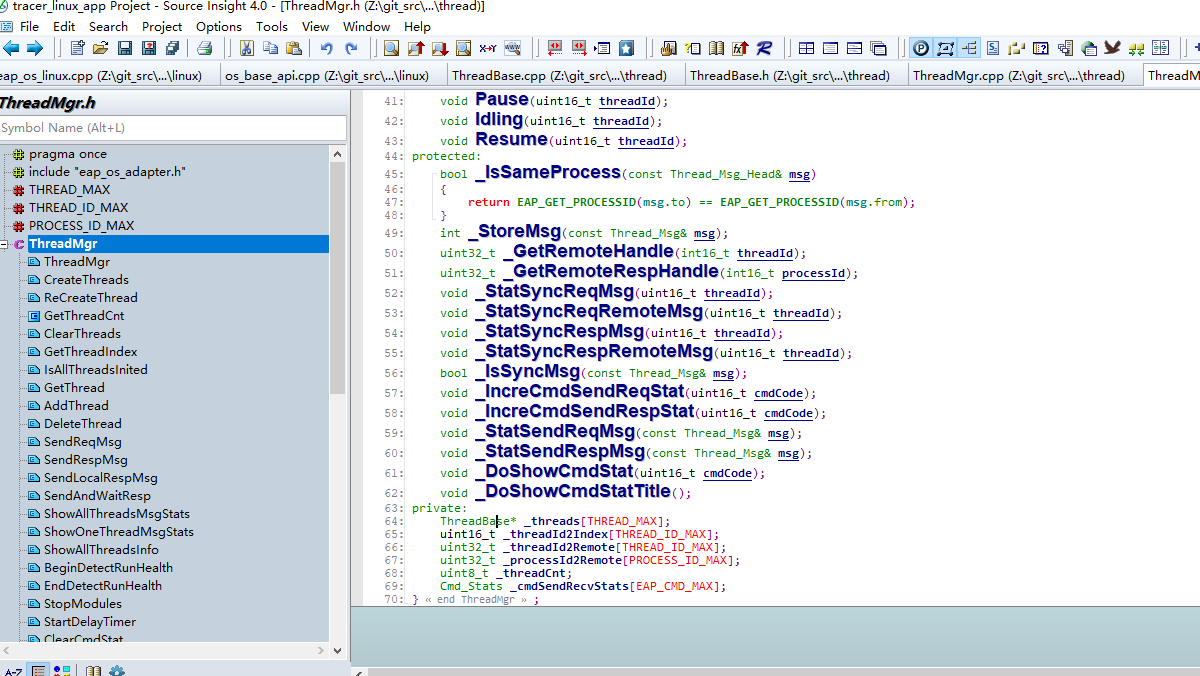
mq\_timedreceive

mq\_receive

## threadMgr.cpp

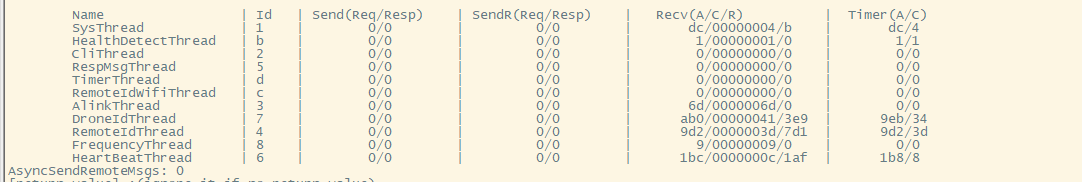
线程管理类，它创建一个数组，管理所有创建的threadBase对象。

1. 每个thread是一个threadbase对象。它在进程内有唯一的threadid宏编号,这是线程的逻辑编号。
2. 从threadid找threadbase对象方法。
3. Threadid🡪thread index(实际启动的thread按顺序编号)🡪thread对象表(以index作索引下标) 为什么要从threadid转成index呢。因为threadid有一部分可能没启动，所以直接以threadid作为索引，thread对象表空间要大很多，而用实际启动的index编号，则表空间要小一点，检索会快一些。

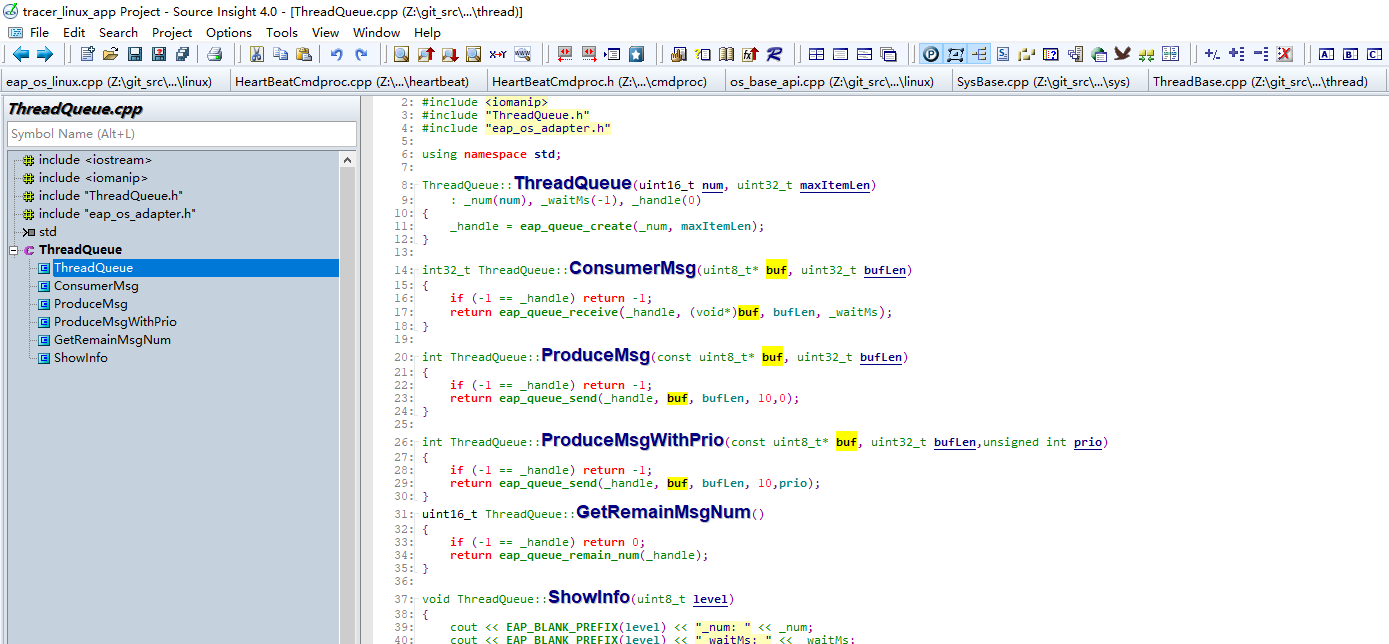


## threadbase.cpp

### ShowMsgStats



## threadQueue.cpp



## 发现的问题

### 跨process的功能没有代码实现

### 一些函数命名与实际功能不附

如：eap\_set\_processid\_for\_remote 实际上应该是eap\_process\_queue\_set

Remote让人不知所去。

Get和set命名不统一等

### 共享内存的信息定义不合理

Thread\_queue\_table, rsp\_process\_queue\_table似乎没有意义。

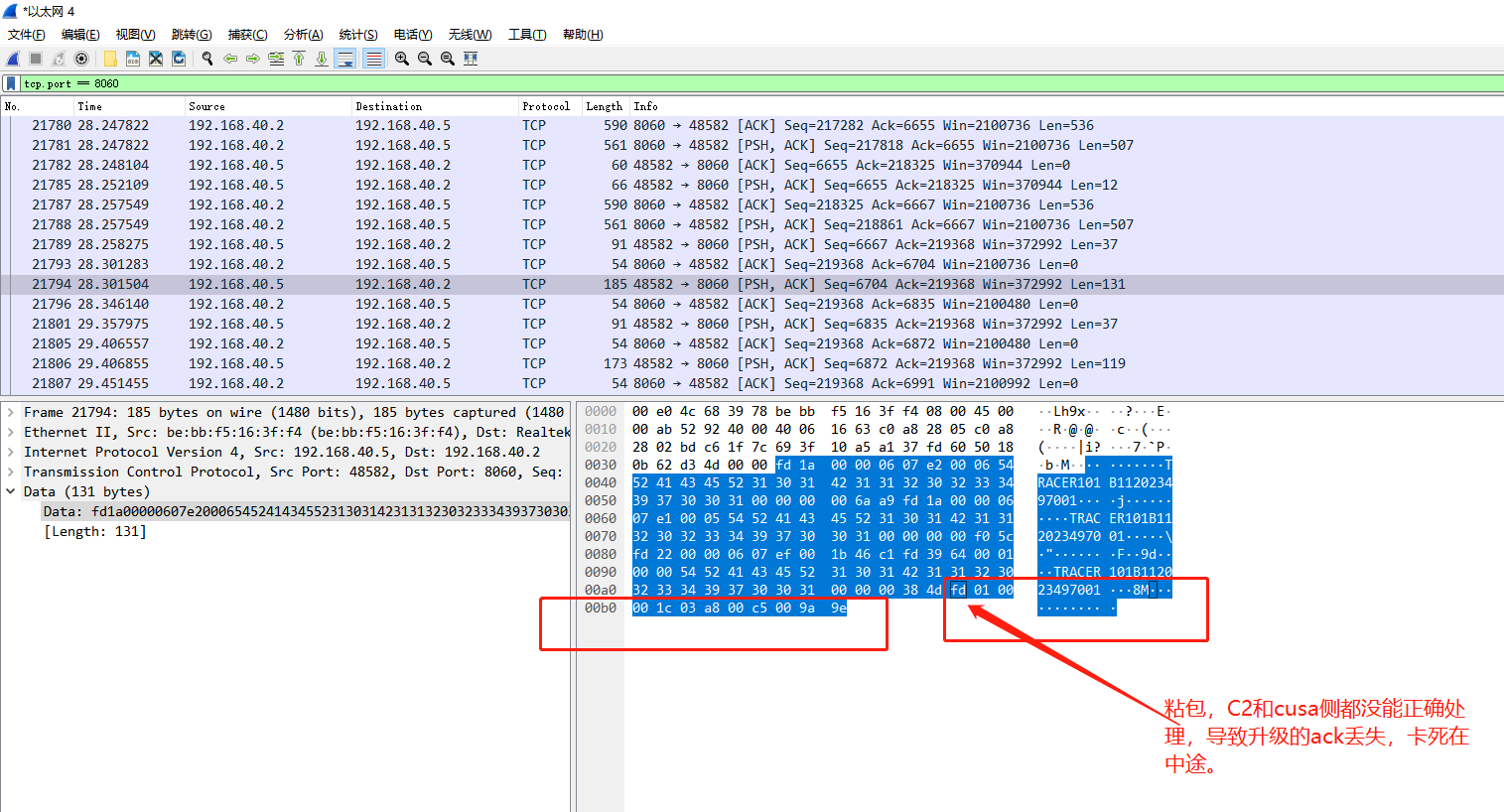
# 定时器机制

启一个线程，专门产生定时器事件，

然后将到时的定时器事件，发消息到相应的模块线程

# 问题

## tcp粘包问题



## Alink消息大小与模块的cmd消息及组包缓冲的问题

现在alink消息是通过模块间消息传递，存在以下问题。

Alink消息为了传升级文件和pil,它希望消息能尽可能大，最大64K+11，这就需要组包缓冲要2倍的消息。

模块消息采用拷贝方式，不可取。它直接附在消息头后面作为数据区，对大部分消息又太大。

使用mq\_open打开消息队列 但是用ipcs看不到 是什么原因

如果您使用mq\_open函数打开一个消息队列，但在使用ipcs命令查看时看不到该消息队列，可能有以下几个原因：

1. 权限问题：mq\_open函数默认创建的消息队列权限为当前用户的umask权限。如果当前用户的umask权限设置得很高，可能导致其他用户无法查看该消息队列。您可以尝试使用mq\_open函数的第三个参数来指定合适的权限，例如0666。
2. 消息队列未命名：mq\_open函数默认创建的消息队列是未命名的，即没有在文件系统中创建对应的文件。因此，使用ipcs命令是无法看到未命名的消息队列的。只有通过mq\_open函数创建的命名消息队列才能在ipcs命令中看到。
3. 消息队列已被删除：如果之前创建的消息队列已经被删除，那么即使使用mq\_open函数重新打开，也无法在ipcs命令中看到。您可以尝试使用mq\_unlink函数删除消息队列，并重新运行程序创建新的消息队列。

请注意，ipcs命令主要用于查看System V IPC（包括消息队列、共享内存和信号量）的信息，而mq\_open函数是用于操作POSIX消息队列的。这两者是不同的机制，因此无法直接通过ipcs命令查看POSIX消息队列。如果您想查看POSIX消息队列的信息，可以使用ls /dev/mqueue命令来查看消息队列的文件名。

## eap\_queue\_send error问题

eap\_queue\_send error, queue\_id=59, length=20, data(hex) 00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00, errno: Connection timed out

# EAP平台功能规化

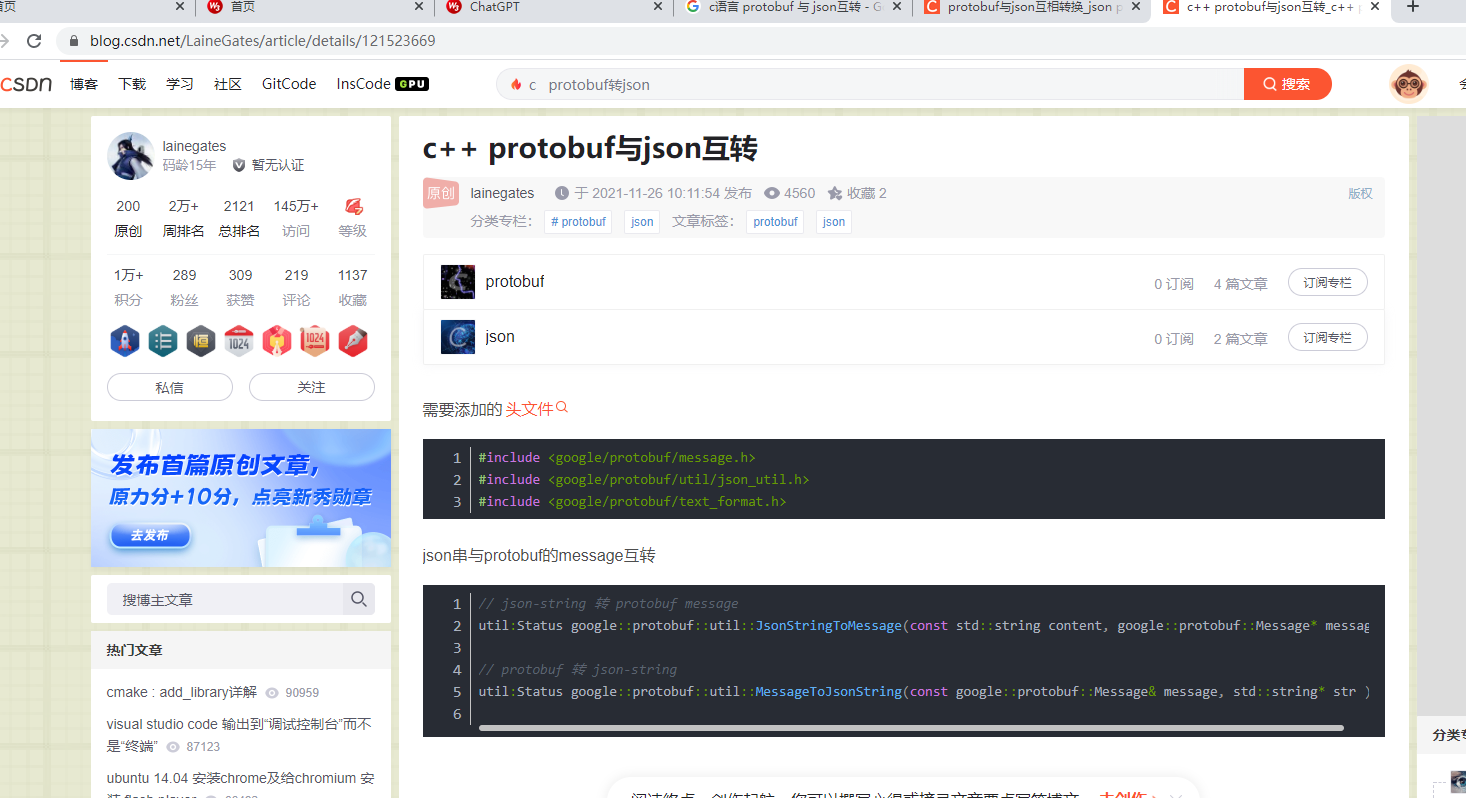
## Pb与json互转接口

<https://github.com/shramov/json2pb>

<https://blog.csdn.net/LaineGates/article/details/121523669>

<https://github.com/protobuf-c/protobuf-c>

protobuf-c可以实现pb与json互转，自动生成了相应的c函数。



将c++的JsonStringToMessage函数 封装成一个c函数

To wrap the JsonStringToMessage function from C++ into a C function, you can create a C header file and implementation file. Here's an example:

**json\_wrapper.h**:

c复制代码

#ifndef JSON\_WRAPPER\_H

#define JSON\_WRAPPER\_H

#ifdef \_\_cplusplus

extern "C" {

#endif

// C function declaration

int json\_string\_to\_message(const char\* json\_string, void\* message);

#ifdef \_\_cplusplus

}

#endif

#endif // JSON\_WRAPPER\_H

**json\_wrapper.cpp**:

cpp复制代码

#include "json\_wrapper.h"

#include <google/protobuf/util/json\_util.h>

#ifdef \_\_cplusplus

extern "C" {

#endif

// C function implementation

int json\_string\_to\_message(const char\* json\_string, void\* message) {

if (json\_string == NULL || message == NULL) {

return -1;

}

// Convert the JSON string to Protobuf message

google::protobuf::util::JsonParseOptions options;

if (!google::protobuf::util::JsonStringToMessage(json\_string, message, options).ok()) {

return -1;

}

return 0;

}

#ifdef \_\_cplusplus

}

#endif

## 硬件信息配置文件pb化

需求：

(1)目前，系统中有多个硬件版本，不同版本，对应的硬件gpio等信息，需要有配套的调整

(2)根据硬件的版本号（从sn中读取），选择相应的硬件配置文件。

(3)linux app启动时，读入硬件配置文件，然后各模块根据所需获取对应的信息进行初始化系统。

## 软件版本属性配置文件pb化

与硬件配置文件类似，提供软件功能上的配置属性

# 结束