## 系统资源

### 信号量

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 创建 | 获取 | 释放 | 删除 | 功能 | 说明 |
| csp\_can\_id\_sem | csp\_can\_id\_init | csp\_can\_id\_get | csp\_can\_id\_get | / | 发送包数据时CAN帧的ID域更新。 | csp\_can\_tx时更新CAN ID的ID位域。由于多线程，id的更新需要保护。也可以使用临界段方式处理。 |
| sport\_lock | csp\_conn\_init | csp\_connect | csp\_connect | / | 创建连接时sport递增。包数据id字段的src，dst更新。更新outgoing\_id.sport和incoming\_id.dport | 由于多线程访问需要保护。 |
| conn\_lock | csp\_conn\_init | csp\_conn\_allocate、csp\_close | csp\_conn\_allocate、csp\_close | / | arr\_conn访问保护 | 多线程对arr\_conn访问需要保护。 |
| conn->rdp.tx\_wait | csp\_rdp\_allocate | csp\_rdp\_connect/csp\_rdp\_send | csp\_rdp\_check\_timeouts/csp\_rdp\_new\_packet | csp\_rdp\_allocate | 用信号量代表是否能发送 | 生产者-消费者模式 |

### 互斥量

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 创建 | 锁定 | 释放 | 删除 |  |  |
| arr\_conn[i].lock | csp\_conn\_init | csp\_conn\_lock | csp\_conn\_unlock |  | 连接锁定与释放 |  |

### 队列

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 队列名字 | 元素 | 长度 | 模块 |  |
| csp\_promisc\_queue | sizeof(csp\_packet\_t \*) | csp\_promisc\_enable函数指定 | csp\_promisc |  |
| qfifo[CSP\_ROUTE\_FIFOS] | csp\_qfifo\_t | CSP\_FIFO\_INPUT | csp\_qfifo | 队列数组  不使能QOS时CSP\_ROUTE\_FIFOS为0，使能时为4 |
| qfifo\_events | sizeof(int) | CSP\_FIFO\_INPUT | 使能QOS有效 |
| conn->rdp.tx\_queue | csp\_packet\_t \* | CSP\_RDP\_MAX\_WINDOW | csp\_rdp | 发送队列，只存储内存地址 |
| conn->rdp.rx\_queue | csp\_packet\_t \* | CSP\_RDP\_MAX\_WINDOW \* 2 | csp\_rdp | 接收队列，只存储内存地址 |
| csp\_buffers | void \* | 用户初始化指定count长度 | csp\_buffer | 只存储指针值 |
| arr\_conn[i].rx\_queue[prio] | csp\_packet\_t \* | CSP\_RX\_QUEUE\_LENGTH | csp\_conn | 只储存内存地址 |
| arr\_conn[i].rx\_event | int | CSP\_CONN\_QUEUE\_LENGTH | csp\_conn | 做消息通知用，所以元素值随意就是0 |
| sock->socket | csp\_packet\_t \* | CSP\_CONN\_QUEUE\_LENGTH | csp\_io.c | 存疑应该是csp\_conn\_t\*  因为csp\_accept时  (csp\_queue\_dequeue(sock->socket, &conn, timeout)出列的时conn |
| socket->socket | csp\_conn\_t \* | csp\_listen时用户指定conn\_queue\_length | csp\_port | 存疑应该是csp\_recvfrom时是  csp\_queue\_dequeue(socket->socket, &packet, timeout);出列的时packet |
|  |  |  |  |  |

### 任务

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 任务名 | 任务函数 | 创建函数 |  |
| 1 | BRIDGE | csp\_bridge | csp\_bridge\_start | 桥接任务 |
| 2 | RTE | csp\_route\_work | csp\_route\_start\_task | 路由处理任务  数据收，转发 |
| 3 | 其他应用层任务 |  |  |  |

### 内存

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 大小 | 创建 | 释放 | 说明 |
| csp\_rtable\_cidr动态路由链表 | sizeof(csp\_rtable\_t) | csp\_rtable\_set | csp\_rtable\_clear | 没创建一条路由需要分配一次 |
| csp\_buffer内存池 | count \* skbfsize; | csp\_buffer\_init | 分配成功后不释放。  队列初始化失败则csp\_buffer\_init立即释放 | 一次分配内存池，条数\*每条大小 |
| csp\_service\_handler | task\_list\_size | csp\_service\_handler | csp\_service\_handler用完立即释放 | ps命令显示任务信息 |
| csp\_sfp.c | sfp\_header->totalsize | csp\_sfp\_recv\_fp | 用户释放 | Sfp接收数据应用使用完后释放 |

### 时基

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 函数 |  |  |  |
| Clock.h | clock\_get\_time  clock\_set\_time | 用户实现该接口 | CSP\_CMP\_CLOCK协议使用do\_cmp\_clock |  |
| Time.h  csp\_time.c | csp\_get\_ms  csp\_get\_ms\_isr  csp\_get\_s  csp\_get\_s\_isr | 用户实现 | 供时间戳等使用 |  |

## 数据流

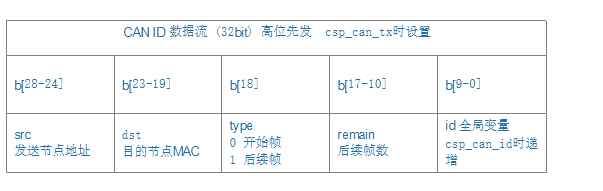
CAN由ID + 8字节的帧形式传输。

底层需要按协议组成包。

### CAN ID

CAN一帧由4字节ID和8字节数据组成。

其中ID组成如下:



其中dst的5位代表目的地址,用于CAN帧过滤。CAN驱动设置该五位必须与本机地址完全匹配，用于帧过滤。

remain是后续CAN帧数。

id是一个全局变量初始化为随机值，会持续递增，代表帧数

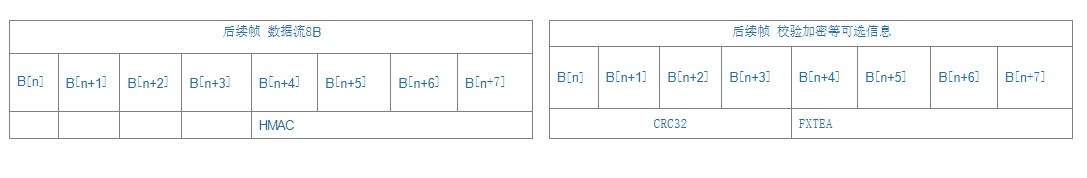
### 包结构

#### 首帧



源端口是全局变量 csp\_conn\_init初始化时随机值,后续递增，代表包数。

#### 后续帧



8字节一帧，后面根据配置添加HMAC，CRC32，FHTEA等信息，由首帧的flags确认是否有这些信息。

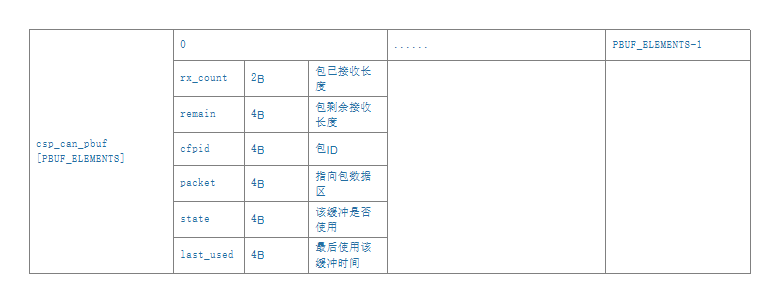
## csp\_if\_can.c(包接收处理)

该文件主要实现获取整包CAN数据。CAN一帧只有8字节。所以需要按照协议将帧组合成包。

### 缓冲区设计

定义了一个大的缓冲区 csp\_can\_pbuf[PBUF\_ELEMENTS]

PBUF\_ELEMENTS设置为 CSP\_CONN\_MAX即 最大连接数,默认为10。



接收到首帧时csp\_can\_pbuf\_new 分配csp\_buffer\_get分配空间存储包数据，

一帧接收完后释放数据csp\_can\_pbuf\_free，并通过csp\_new\_packet写入缓冲区。

### CAN ID过滤

见csp\_can\_init函数说明

### 帧顺序乱Out of order MORE frame received

每个包的缓冲区与CAN ID绑定 csp\_can\_pbuf\_find匹配CFP\_ID\_CONN\_MASK

如果当前CAN ID在缓冲区找不到，且当前帧不是首帧说明帧乱了。

正常只有首帧的CAN ID在缓冲区才没有，后续帧一定能在缓冲区找到。

### 未完成帧Incomplete frame

如果新接收到首帧，能找到ID对应的缓冲区，说明前面有帧未接收完。

### 首帧太短Short BEGIN frame received

如果首帧的长度小于 首帧头 ID + LEN域的6字节时，说明帧错误。

### 帧丢失CAN frame lost in CSP packet

如果当前帧里的remain域 缓冲区里记录的值不匹配，说明有帧丢失。

### 包超时清理csp\_can\_pbuf\_cleanup

如果多次未接收到CAN帧，则调用csp\_can\_pbuf\_cleanup进行包清除处理。

如果有缓冲区正在使用，且对应的最后使用时间与当前时间超过PBUF\_TIMEOUT\_MS

则调用csp\_can\_pbuf\_free(buf)释放该缓冲区。PBUF\_TIMEOUT\_MS默认为10秒。

if (now - buf->last\_used > PBUF\_TIMEOUT\_MS) {

csp\_log\_warn("CAN Buffer element timed out");

/\* Recycle packet buffer \*/

csp\_can\_pbuf\_free(buf);

}

### 数据结构

csp\_queue\_handle\_t

### 变量

csp\_can\_id\_sem id访问信号量

csp\_can\_id id值

csp\_can\_pbuf\_element\_t 缓冲区数组

### 宏

略

### 函数

#### int csp\_can\_init(uint8\_t mode, struct csp\_can\_config \*conf)

初始化缓冲区csp\_can\_pbuf\_init

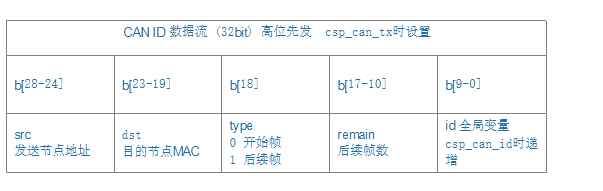
初始化id csp\_can\_id\_init

获取 CAN ID掩码 即匹配CANID 的dst位域

mask = CFP\_MAKE\_DST((1 << CFP\_HOST\_SIZE) - 1);

can\_init(CFP\_MAKE\_DST(csp\_get\_address()), mask, conf) != 0)

设置CAN底层过滤。



csp\_iflist\_add(&csp\_if\_can);

将can接口加到接口表里去。

csp\_iface\_t csp\_if\_can = {

.name = "CAN",

.nexthop = csp\_can\_tx,

.mtu = CSP\_CAN\_MTU,

};

此时接口和设备地址还没对应，csp\_route\_set函数设置后设备地址和接口才绑定。

这样才知道哪个地址对应哪个接口发出去。

#### csp\_can\_id\_init

初始化全局变量csp\_can\_id 为随机值，并创建访问该变量的信号量。该值在发送数据包时csp\_can\_tx递增，附带在CAN ID的字后10bit。一包数据的没帧该值不变。

发送下一包时才改变。

这样接收端可以根据该值，去判断帧是否属于一帧，csp\_can\_pbuf\_find也要匹配该位域。

#### csp\_can\_id\_get

csp\_can\_tx发送包时调用该函数获取CAN ID的bit0-bit9。没发送一包增加1。因为多任务访问，所以使用信号量保护。

#### csp\_can\_pbuf\_init

初始化全局数组csp\_can\_pbuf，此时packet还未分配内存，接收到首帧时才分配。

#### csp\_can\_pbuf\_timestamp

设置当前ms时间戳到缓冲区，通过clock.c的csp\_get\_ms();获取ms值，需要用户实现。

#### static int csp\_can\_pbuf\_free(csp\_can\_pbuf\_element\_t \*buf)

释放一个缓冲区，如果packet 指向了有效内存，则要调用

csp\_buffer\_free(buf->packet);释放内存。

#### static csp\_can\_pbuf\_element\_t \*csp\_can\_pbuf\_new(uint32\_t id)

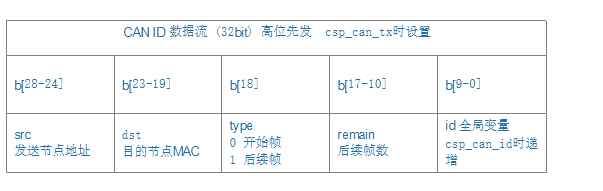
分配一个缓冲区，将id值写入，就可以根据id值进行匹配。

并设置BUF\_USED状态和时间戳。此时packet没有立即分配内存。

在后面判断为有效帧才分配。

#### static csp\_can\_pbuf\_element\_t \*csp\_can\_pbuf\_find(uint32\_t id, uint32\_t mask)

查找指定的ID在缓冲区是否已经分配，匹配如下的 src dst和id位域。



如果匹配则更新时间戳。

#### void csp\_can\_pbuf\_cleanup(void)

如果长时间无新的CAN帧收到，则调用该函数进行清理。 见”包超时处理”

#### int csp\_can\_process\_frame(can\_frame\_t \*frame)

包接收处理 通过该函数实现

1.首先根据CAN ID查找，是否有对应的缓冲区

如果有对应ID的缓冲区没有，且当前帧是首帧则 则要申请缓冲区csp\_can\_pbuf\_new

如果有对应ID的缓冲区没有，且当前帧不是首帧 出错Out of order MORE frame received

1. 如果对应ID的缓冲区有数据buf->packet != NULL，且当前帧是首帧 则出错Incomplete frame

3. 如果有对应ID的缓冲区没有，且当前帧是首帧

或者 如果有对应ID的缓冲区有，且当前帧不是首帧 才是有效帧

3.1 首帧处理

检查长度如果小于6字节 说明帧长度太短无效 Short BEGIN frame received

分配内存csp\_buffer\_get(CSP\_CAN\_MTU)存储数据

获取首帧数据的id和len等值，并且转化为主机序，buf->rx\_count接收数清0，offset偏移6字节头，记录剩余帧数buf->remain

3.2 后续帧处理

检查CAN ID的remian和记录的buf->remain是否匹配。

检查buf->rx\_count + frame->dlc - offset) > buf->packet->length) 是否溢出。

检查是否接收完buf->rx\_count != buf->packet->length

如果接收完csp\_new\_packet csp\_qfifo\_write 将packet发送到到qfifo队列

释放csp\_can\_pbuf\_free

/\* Drop packet buffer reference \*/

buf->packet = NULL;

/\* Free packet buffer \*/

csp\_can\_pbuf\_free(buf);

buf->packet = NULL;置空，所以csp\_can\_pbuf\_free(buf);不会释放buf->packet,而是将packet指针发送到了队列。

#### int csp\_can\_tx(csp\_iface\_t \*interface, csp\_packet\_t \*packet, uint32\_t timeout)

包发送处理

计算CAN ID 其中id部分使用csp\_can\_id\_get获取(发送一包 递增1) dest则为查找路由表csp\_rtable\_find\_mac目标地址对应的MAC ，没有MAC则设置为目的地址

先发送首帧 再连续发送后续帧。

### 总结分析

1. 接收缓冲区溢出不是跟 缓冲区大小比较

if ((buf->rx\_count + frame->dlc - offset) > buf->packet->length) {

csp\_log\_error("RX buffer overflow");

csp\_if\_can.frame++;

csp\_can\_pbuf\_free(buf);

break;

}

因为分配内存大小为buf->packet = csp\_buffer\_get(CSP\_CAN\_MTU);

所以接收的数据长度不能超过CSP\_CAN\_MTU。要检查不能超过CSP\_CAN\_MTU。

1. 如果发送的实际数据和buf->packet->length不相同 则buf->rx\_count != buf->packet->length可能一直为真。
2. 发送的实际数据和buf->packet->length大于CSP\_CAN\_MTU

则会溢出。

因为如果设置的buf->packet->length大于CSP\_CAN\_MTU则会溢出。(实测 客户端发送300字节溢出CSP\_CAN\_MTU为256时 )

if (buf->rx\_count != buf->packet->length)

和if ((buf->rx\_count + frame->dlc - offset) > buf->packet->length)

均不会满足

改为

if(buf->packet->length > CSP\_CAN\_MTU)

{

csp\_log\_error("RX too long");

csp\_if\_can.frame++;

csp\_can\_pbuf\_free(buf);

break;

}

/\* Check for overflow \*/

if (((buf->rx\_count + frame->dlc - offset) > buf->packet->length) || ((buf->rx\_count + frame->dlc - offset) > CSP\_CAN\_MTU)) {

csp\_log\_error("RX buffer overflow");

csp\_if\_can.frame++;

csp\_can\_pbuf\_free(buf);

break;

}

## csp\_buffer

### 变量

static csp\_queue\_handle\_t csp\_buffers;

static char \* csp\_buffer\_pool;

static unsigned int count, size;

csp\_buffers:一个队列,用于记录哪些块分配了，哪些块没有分配。初始化时将所有块入列,分配内存时出列，释放块时入列。

csp\_buffer\_pool :malloc出来的一大块内存，块分配就是从这块内存里分配。

count块数

size 每块的用户数据字节+csp\_packet\_t结构体头部大小

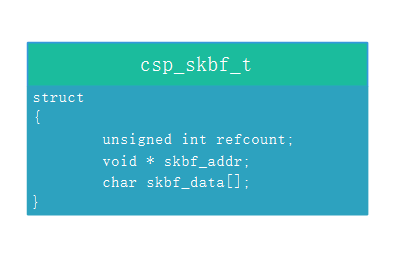
csp\_buffer\_pool :malloc分配出来的内存大于count\*size因为还要存储控制字段等。

### 宏

CSP\_BUFFER\_ALIGN 分配内存的对齐大小

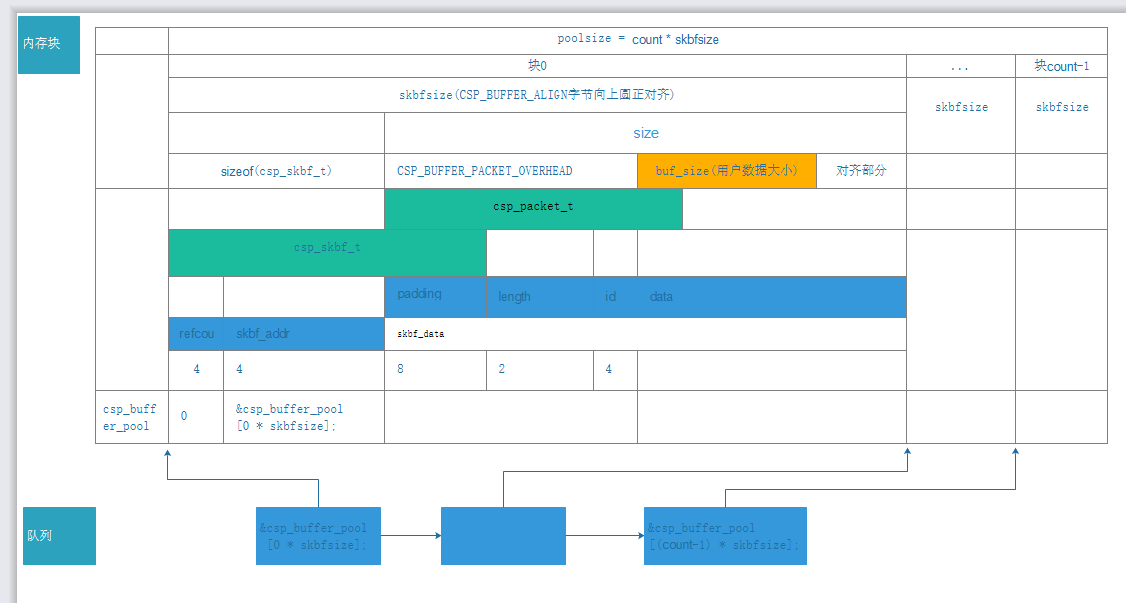
### 数据结构

csp\_skbf\_t



### 函数

#### csp\_buffer\_init



int csp\_buffer\_init(int buf\_count, int buf\_size) {

unsigned int i;

csp\_skbf\_t \* buf;

count = buf\_count;

size = buf\_size + CSP\_BUFFER\_PACKET\_OVERHEAD;

unsigned int skbfsize = (sizeof(csp\_skbf\_t) + size);

skbfsize = CSP\_BUFFER\_ALIGN \* ((skbfsize + CSP\_BUFFER\_ALIGN - 1) / CSP\_BUFFER\_ALIGN);

unsigned int poolsize = count \* skbfsize;

csp\_buffer\_pool = csp\_malloc(poolsize);

if (csp\_buffer\_pool == NULL)

goto fail\_malloc;

csp\_buffers = csp\_queue\_create(count, sizeof(void \*));

if (!csp\_buffers)

goto fail\_queue;

if (CSP\_INIT\_CRITICAL(csp\_critical\_lock) != CSP\_ERR\_NONE)

goto fail\_critical;

memset(csp\_buffer\_pool, 0, poolsize);

for (i = 0; i < count; i++) {

/\* We have already taken care of pointer alignment since

\* skbfsize is an integer multiple of sizeof(int \*)

\* but the explicit cast to a void \* is still necessary

\* to tell the compiler so.

\*/

buf = (void \*) &csp\_buffer\_pool[i \* skbfsize];

buf->refcount = 0;

buf->skbf\_addr = buf;

csp\_queue\_enqueue(csp\_buffers, &buf, 0);

}

return CSP\_ERR\_NONE;

fail\_critical:

csp\_queue\_remove(csp\_buffers);

fail\_queue:

csp\_free(csp\_buffer\_pool);

fail\_malloc:

return CSP\_ERR\_NOMEM;

}

初始化过程:

1. 先分配poolsize大小的内存,poolsize = count \* skbfsize;指针为csp\_buffer\_pool。

skbfsize为csp\_skbf\_t结构体头部(不含skbf\_data区域) + csp\_packet\_t结构头部(不含data区域),按照CSP\_BUFFER\_ALIGN对齐。

1. 创建队列csp\_queue\_create。
2. 将分配的内存csp\_buffer\_pool清0。
3. 将每个块的地址&csp\_buffer\_pool[i \* skbfsize]值,写入队列,并设置refcount域为0代表没有分配,skbf\_addr域就设置为该块的地址&csp\_buffer\_pool[i \* skbfsize]。

#### csp\_buffer\_get、csp\_buffer\_get\_isr

获取内存即从队列中出列存储块地址的值。该值强制转换成csp\_skbf\_t \* buffer类型。

Buffer->skbf\_addr即该块地址。

取出后buffer->refcount++;表示已经分配。

队列中获取的值实际就是块地址,块的第二个区域skbf\_addr也是存的快地址,这样可以做交叉验证buffer != buffer->skbf\_addr确保内存未被破坏。

csp\_buffer\_get\_isr与csp\_buffer\_get实现时一样的,只是底层调用的队列API相应也是ISR可调用API。另外csp\_buffer\_get\_isr中无日志输出。

#### csp\_buffer\_free、csp\_buffer\_free\_isr

void csp\_buffer\_free(void \*packet)释放 packet地址对应的块，

csp\_skbf\_t \* buf = (csp\_skbf\_t \*)((uint32\_t)packet - sizeof(csp\_skbf\_t));

先偏移到块开始地址,

然后校验地址buf % CSP\_BUFFER\_ALIGN是不是对齐

校验buf->skbf\_addr != buf地址是不是一致

buf->refcount == 0 或者buf->refcount > 1 是不是只对分配了一次的内存释放。

最后将buf值入列。

csp\_buffer\_free\_isr实现与csp\_buffer\_free是一样的,只是底层调用的队列API相应也是ISR可调用API。另外csp\_buffer\_free\_isr中无日志输出。

#### csp\_buffer\_clone

void \*csp\_buffer\_clone(void \*buffer)

分配内存块用于存储buffer对应的包数据

csp\_packet\_t \*clone = csp\_buffer\_get(packet->length);获取块内存

然后将memcpy(clone, packet, size);

数据拷贝到新分配的块,并返回块地址。

#### csp\_buffer\_remaining

获取可分配块数,即队列里可用的项数。

csp\_queue\_size(csp\_buffers)

#### csp\_buffer\_size

获取每块数据大小,除了内存块控制区域refcount和skbf\_addr以外的部分。

### 总结分析:

1. 可以移植性不是很好，使用了unsigned int等数据类型。
2. 使用了内存地址到结构体指针的强制转换，需要注意是否在不同平台编译器会有问题。
3. 依赖系统的队列操作,可优点是直接支持多线程,缺点是需要系统支持,可以移植性降低。
4. csp\_skbf\_t使用了0长数组可移植性不好。
5. 使用注意:
   1. 需定义CSP\_BUFFER\_ALIGN 默认是(sizeof(int \*))
   2. csp\_buffer\_init初始化时要保证,初始化分配的buf\_size要大于实际最大的packet的大小。

## csp\_qfifo.c

### 数据结构

typedef struct {

csp\_iface\_t \* interface;

csp\_packet\_t \* packet;

} csp\_qfifo\_t;

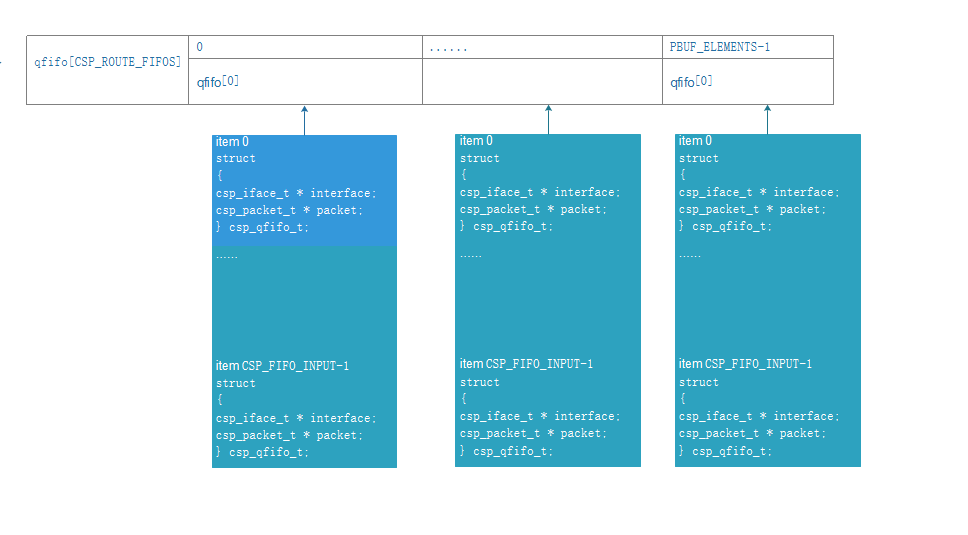
### 变量

static csp\_queue\_handle\_t qfifo[CSP\_ROUTE\_FIFOS];

CSP\_ROUTE\_FIFOS默认为1，可配置。

static csp\_queue\_handle\_t qfifo\_events;

### 缓冲区设计



定义了一个消息队列数组，如果不使能优先级，则数组大小为1，即CSP\_ROUTE\_FIFOS=1

如果使能了优先级则CSP\_USE\_QOS=(1 << CSP\_ID\_PRIO\_SIZE)=4 即优先级0-3，一个优先级对应一个队列。

每个队列长度为CSP\_FIFO\_INPUT 10

队列的元素sizeof(csp\_qfifo\_t)大小是sizeof(csp\_qfifo\_t)。

所以队列值存储了接收到的packet数据的指针和对应的接口指针。Packet是csp\_can\_process\_frame时分配的。

### 函数

#### csp\_qfifo\_init

创建指定个数队列qfifo[prio] = csp\_queue\_create(CSP\_FIFO\_INPUT, sizeof(csp\_qfifo\_t));

如果使能了优先级还要创建通知队列 qfifo\_events = csp\_queue\_create(CSP\_FIFO\_INPUT, sizeof(int));

#### int csp\_qfifo\_read(csp\_qfifo\_t \* input)

如果使能了优先级，则先读通知队列qfifo\_events

然后再读最高优先级的qfifo队列 qfifo[0]优先级最高

如果没有优先级则 从队列qfif[0]读出数据 if (csp\_queue\_dequeue(qfifo[0], input, FIFO\_TIMEOUT) != CSP\_QUEUE\_OK)

#### void csp\_qfifo\_write(csp\_packet\_t \* packet, csp\_iface\_t \* interface, CSP\_BASE\_TYPE \* pxTaskWoken)

往队列写数据result = csp\_queue\_enqueue(qfifo[fifo], &queue\_element, 0);

如果使能了优先级 还要写通知队列csp\_queue\_enqueue(qfifo\_events, &event, 0);

### 分析总结

1. CSP\_USE\_QOS=1时

Qfifo入列有结果检查，但是qfifo\_event则没有检查

if (result == CSP\_QUEUE\_OK) {

if (pxTaskWoken == NULL)

csp\_queue\_enqueue(qfifo\_events, &event, 0);

else

csp\_queue\_enqueue\_isr(qfifo\_events, &event, pxTaskWoken);

}

1. 使能QOS时

每个优先级对应队列长度为CSP\_FIFO\_INPUT

qfifo[prio] = csp\_queue\_create(CSP\_FIFO\_INPUT, sizeof(csp\_qfifo\_t));

通知队列长度也是CSP\_FIFO\_INPUT

qfifo\_events = csp\_queue\_create(CSP\_FIFO\_INPUT, sizeof(int));

如果每个优先级都来CSP\_FIFO\_INPUT个数据，则qfifo\_events将无法全部保存，所以要设置为CSP\_FIFO\_INPUT\*CSP\_PRIORITIES长，没有一个数据来都能产生qfifo\_events通知。

改为qfifo\_events = csp\_queue\_create(CSP\_FIFO\_INPUT\*CSP\_PRIORITIES, sizeof(int));

## csp\_iflist

### 变量

Interfaces 接口链表

### 宏

### 数据结构

### 函数

#### csp\_iflist\_get\_by\_name

根据接口名字字符串遍历链表，返回接口节点

#### void csp\_iflist\_add(csp\_iface\_t \*ifc)

在链表末尾插入接口节点,如果已经插入则不再插入

#### void csp\_iflist\_print(void)

打印接口信息

### 总结分析:

1. csp\_iflist\_print中使用了浮点运算,正式版本不要使能调试信息
2. 接口名字 name不要超过10个字符,因为匹配只比较10个字符长度。

## csp\_rtable\_static.c

静态路由表

用结构体数组存储路由信息

static csp\_rtable\_t routes[CSP\_ROUTE\_COUNT] = {0};

CSP\_ROUTE\_COUNT = (CSP\_ID\_HOST\_MAX + 2) = ((1 << (CSP\_ID\_HOST\_SIZE)) - 1) + 2

= ((1 << (5)) - 1) + 2 = 33

地址即作为数组索引,缺点是固定分配浪费空间。

CSP\_DEFAULT\_ROUTE = (CSP\_ID\_HOST\_MAX + 1)=32 默认设备地址 回环处理

CSP\_BROADCAST\_ADDR =CSP\_ID\_HOST\_MAX=31 广播地址 即0x1F

即5位数最大表示的数,地址阈占用5bit。

初始化时会默认设置CSP\_DEFAULT\_ROUTE的路由信息，接口为csp\_if\_lo

## csp\_rtable\_cidr

动态链表管理

### 变量

static csp\_rtable\_t \* rtable = NULL;

路由链表

### 宏

### 数据结构

typedef struct \_\_attribute\_\_((\_\_packed\_\_)) csp\_rtable\_s {

uint8\_t address;

uint8\_t netmask;

uint8\_t mac;

csp\_iface\_t \* interface;

struct csp\_rtable\_s \* next;

} csp\_rtable\_t;

### 函数

#### csp\_rtable\_find

根据地地址和掩码查找路由,可指定精确匹配还是掩码匹配

static csp\_rtable\_t \* csp\_rtable\_find(uint8\_t addr, uint8\_t netmask, uint8\_t exact)

exact =1 必须addr和netmas完全一致

exact=0 则只匹配高netmask位

dddr是5位大小,如果netmask=5则表示高5位都参与匹配，就是完全匹配。

如果netmask=1则表示高1位参与匹配,低四位忽略。

#### csp\_rtable\_clear

遍历释放动态分配的路由，然后设置默认

路由csp\_rtable\_set(csp\_get\_address(), CSP\_ID\_HOST\_SIZE, &csp\_if\_lo, CSP\_NODE\_MAC);

#### csp\_rtable\_parse

解析字符串的address, netmask, ifc, mac，支持字符串中多组有效值。

#### csp\_rtable\_load

根据字符串值设置路由表

调用csp\_rtable\_parse(buffer, 0);实现

#### csp\_rtable\_check

检查字符串对应的接口是否存在

调用csp\_rtable\_parse(buffer, 1);实现

#### csp\_rtable\_save

保存路由表到缓冲区，如果mac地址不是CSP\_NODE\_MAC

则保存mac地址 如下

i->address, i->netmask, i->interface->name, i->mac

否则保存

1. >address, i->netmask, i->interface->name

#### csp\_iface\_t \* csp\_rtable\_find\_iface(uint8\_t id)

根据地址查找接口，匹配5位地址

#### uint8\_t csp\_rtable\_find\_mac(uint8\_t id)

根据地址查找mac，匹配5位地址

#### int csp\_rtable\_set(uint8\_t \_address, uint8\_t \_netmask, csp\_iface\_t \*ifc, uint8\_t mac)

设置路由表

如果默认地址位CSP\_DEFAULT\_ROUTE默认路由，则地址和掩码都设置为0

netmask = 0;

address = 0;

然后查找路由是否存在csp\_rtable\_find(address, netmask, 1);

如果不存在则分配新的路由，添加到路由链表末尾，并填充新设置的路由信息。

### 总结:

1. csp\_rtable\_find函数

uint8\_t best\_result\_mask = 0; netmask为uint8\_t类型

所以i->netmask >= best\_result\_mask永远为真 该语句无意义，best\_result\_mask也没什么用。

1. csp\_rtable\_parse函数

未对解析出的address, netmask, ifc, mac做校验，

buffer未做长度限制，name[100] 分配100字节占用栈较多。

strtok非线程安全。

## csp\_bridge.c

### 变量

static csp\_iface\_t\* if\_a = NULL;

static csp\_iface\_t\* if\_b = NULL;

### 函数

#### int csp\_bridge\_start(unsigned int task\_stack\_size, unsigned int task\_priority, csp\_iface\_t \* \_if\_a, csp\_iface\_t \* \_if\_b)

创建BRIDGE任务， if\_a = \_if\_a; if\_b = \_if\_b;

桥接\_if\_a 和\_if\_b

#### CSP\_DEFINE\_TASK(csp\_bridge)

桥接任务处理，csp\_qfifo\_read如果读到包数据，如果使能CSP\_USE\_PROMISC，csp\_promisc\_add添加读到队列。

csp\_iface\_t \* ifout;

if (input.interface == if\_a) {

ifout = if\_b;

} else {

ifout = if\_a;

}

如果读到的接口是a，则发送接口是b，否则是a?

这里逻辑是否有问题? 如果读到的既不是a又不是b接口怎么处理?

读到的不是a就全部转发给b?

最后释放csp\_buffer\_free内存。注:csp\_promisc\_add中是将packet克隆出去了。

### 总结分析

1.

CSP\_DEFINE\_TASK(csp\_bridge)，桥接时逻辑

csp\_iface\_t \* ifout;

if (input.interface == if\_a) {

ifout = if\_b;

} else {

ifout = if\_a;

}

不对，

在接收到端口不是if\_a都会转到B。

2.在使用了桥接时，csp\_qfifo\_read将数据读走了，csp\_route\_work有可能读不到，用户将接收不到数据。

## csp\_promisc

### 变量

static csp\_queue\_handle\_t csp\_promisc\_queue = NULL;

static int csp\_promisc\_enabled = 0;

### 函数

#### int csp\_promisc\_enable(unsigned int buf\_size)

创建长度为buf\_size 元素大小为sizeof(csp\_packet\_t \*)的队列csp\_promisc\_queue

设置csp\_promisc\_enabled = 1;

#### void csp\_promisc\_disable(void)

csp\_promisc\_enabled = 0;

#### csp\_packet\_t \* csp\_promisc\_read(uint32\_t timeout)

从csp\_promisc\_queue出列，用户调用该函数读包地址值，用完后需要释放内存。

#### void csp\_promisc\_add(csp\_packet\_t \* packet)

克隆包\*packet\_copy = csp\_buffer\_clone(packet);

并将包地址，写入队列

csp\_queue\_enqueue(csp\_promisc\_queue, &packet\_copy, 0)。

### 总结分析

1. 用户csp\_packet\_t \* csp\_promisc\_read(uint32\_t timeout)调用该函数读 使用后需要释放

packet。

## csp\_dedup 重复判断

#define CSP\_DEDUP\_COUNT 16 判断缓冲池大小

#define CSP\_DEDUP\_WINDOW\_MS 1000 crc校验数据相同 且时间间隔小于该值认为是重复数据。

static uint32\_t csp\_dedup\_array[CSP\_DEDUP\_COUNT] = {0};

每个数据的crc校验

static uint32\_t csp\_dedup\_timestamp[CSP\_DEDUP\_COUNT] = {0};

每个数据的时间戳

static int csp\_dedup\_in = 0; 遍历索引，新来一个数据递增，对CSP\_DEDUP\_COUNT取余

### bool csp\_dedup\_is\_duplicate(csp\_packet\_t \*packet)

计算新数据的crc，遍历缓冲池，如果缓冲池有数据和新数据crc一样，且时间差小于

CSP\_DEDUP\_WINDOW\_MS认为是重复数据。

更新缓冲池数据crc和时间戳，索引更新。

csp\_dedup\_array[csp\_dedup\_in] = crc;

csp\_dedup\_timestamp[csp\_dedup\_in] = csp\_get\_ms();

csp\_dedup\_in = (csp\_dedup\_in + 1) % CSP\_DEDUP\_COUNT;

## csp\_route

路由处理任务，数据接收由该任务处理。

见流程图

## csp\_conn

### 变量

static csp\_conn\_t arr\_conn[CSP\_CONN\_MAX]; 连接数组

static csp\_bin\_sem\_handle\_t conn\_lock; 连接访问信号量

static uint8\_t sport; 源端口

csp\_conn\_init初始化为随机值，创建csp\_connect时递增。

static csp\_bin\_sem\_handle\_t sport\_lock; 源端口访问信号量

### 数据结构

struct csp\_conn\_s {

csp\_conn\_type\_t type; /\* Connection type (CONN\_CLIENT or CONN\_SERVER) \*/

csp\_conn\_state\_t state; /\* Connection state (SOCKET\_OPEN or SOCKET\_CLOSED) \*/

csp\_mutex\_t lock; /\* Connection structure lock \*/

csp\_id\_t idin; /\* Identifier received \*/

csp\_id\_t idout; /\* Identifier transmitted \*/

#ifdef CSP\_USE\_QOS

csp\_queue\_handle\_t rx\_event; /\* Event queue for RX packets \*/

#endif

csp\_queue\_handle\_t rx\_queue[CSP\_RX\_QUEUES]; /\* Queue for RX packets \*/

csp\_queue\_handle\_t socket; /\* Socket to be "woken" when first packet is ready \*/

uint32\_t timestamp; /\* Time the connection was opened \*/

uint32\_t opts; /\* Connection or socket options \*/

#ifdef CSP\_USE\_RDP

csp\_rdp\_t rdp; /\* RDP state \*/

#endif

};

## csp\_udp

void csp\_udp\_new\_packet(csp\_conn\_t \* conn, csp\_packet\_t \* packet)

往conn里的rx队列写入，包地址

csp\_conn\_enqueue\_packet(conn, packet) < 0)

if (conn->socket != NULL)

Socket里入列conn

if (conn->socket != NULL) {

if (csp\_queue\_enqueue(conn->socket, &conn, 0) != CSP\_QUEUE\_OK) {

csp\_log\_warn("Warning socket connection queue full");

csp\_close(conn);

return;

}

与

if (opts & CSP\_SO\_CONN\_LESS) {

sock->socket = csp\_queue\_create(CSP\_CONN\_QUEUE\_LENGTH, sizeof(csp\_packet\_t \*));

if (sock->socket == NULL)

return NULL;

} else {

sock->socket = NULL;

}

创建的内容不一样，是不是应该是 csp\_queue\_create(CSP\_CONN\_QUEUE\_LENGTH, sizeof(csp\_conn\_t\*)); 虽然都是指针实际结果是一样，但是意义不一样。

## csp\_clock.h

需要实现时间基准,

freertos实现如下:

FreeRTOSConfig.h中

#define configUSE\_TICK\_HOOK 1

Clock.c中

csp\_timestamp\_t g\_clock\_t = {(uint32\_t)0,(uint32\_t)0};

void vApplicationTickHook()

{

g\_clock\_t.tv\_nsec += (uint32\_t)1000000 / configTICK\_RATE\_HZ;

while(g\_clock\_t.tv\_nsec >= (uint32\_t)1000000)

{

g\_clock\_t.tv\_nsec -= (uint32\_t)1000000;

g\_clock\_t.tv\_sec++;

}

}

void clock\_get\_time(csp\_timestamp\_t \* time)

{

time->tv\_nsec = g\_clock\_t.tv\_nsec;

time->tv\_sec = g\_clock\_t.tv\_sec;

}

void clock\_set\_time(csp\_timestamp\_t \* time)

{

g\_clock\_t.tv\_nsec = time->tv\_nsec;

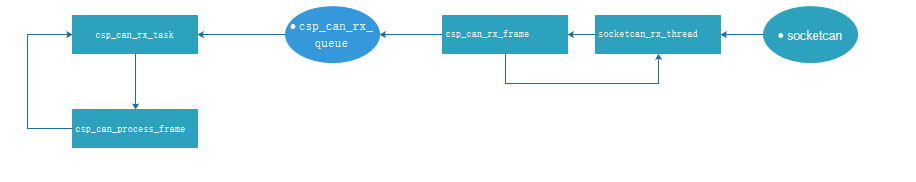
g\_clock\_t.tv\_sec = time->tv\_sec;

}

## CAN链路移植

### 接收流程

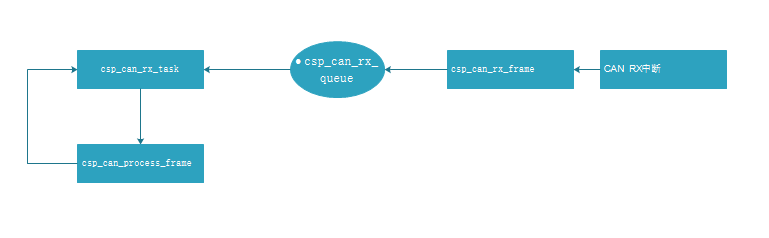
#### 接收流程(linux平台参考例子)



socketcan\_rx\_thread线程循环通过socketcan不断获取帧数据,发送到队列csp\_can\_rx\_queue，

csp\_can\_rx\_task线程不断查询队列，调用csp\_can\_process\_frame进行真正的处理。

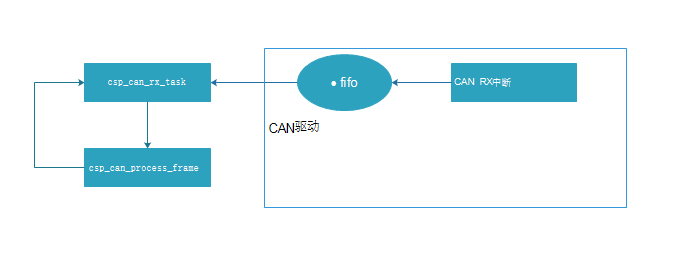
#### 中断发送队列(单片机平台可选方案)



CAN\_RX中断中将帧数据写入队列，csp\_can\_rx\_task线程不断查询队列，调用csp\_can\_process\_frame进行真正的处理。

该方法不需要在CAN驱动中实现缓冲,依赖系统提供的队列进行缓冲。

#### 驱动实现缓冲(单片机平台可选方案)



该方法由CAN驱动实现缓冲，CAN中断中将帧写入缓冲,csp\_can\_rx\_task线程不断查询队列，调用csp\_can\_process\_frame进行真正的处理。

该方法不依赖系统队列API,由驱动实现缓冲。

### 接口移植csp\_if\_can

#### int csp\_can\_init(uint8\_t mode, struct csp\_can\_config \*conf)函数

只保留csp\_can\_pbuf\_init

csp\_can\_id\_init

csp\_iflist\_add

can\_init

四个函数

其中can\_init有一点很重要就是,初始化波特率，另外设置过滤。

设置Id的bit[19-23]要与本机地址匹配。也就是网络上的包目的地址要是本机才会接收。

疑问广播包怎么处理？

广播是在包的packet->id.dst != CSP\_BROADCAST\_ADDR域进行比对。

如果是CAN包如果设置了过滤，还是需要CAN ID的id域匹配才能接收。

或者要设置can驱动为CSP\_CAN\_PROMISC模式不过滤。

static CSP\_DEFINE\_TASK(csp\_can\_rx\_task)任务不需要

csp\_can\_rx\_frame函数不需要

csp\_can\_rx\_queue = csp\_queue\_create(CSP\_CAN\_RX\_QUEUE\_SIZE, sizeof(can\_frame\_t));不需要

ret = csp\_thread\_create(csp\_can\_rx\_task, "CAN", 6000/sizeof(int), NULL, 3, &csp\_can\_rx\_task\_h);不需要

static csp\_thread\_handle\_t csp\_can\_rx\_task\_h;

static csp\_queue\_handle\_t csp\_can\_rx\_queue;

不需要

创建一个任务尽心接收处理

static void can\_rxtask( void \*pvParameters )

{

kiss\_set(0);

while(1)

{

can\_rx\_handle(0);

//driver\_can\_sendloop();

OsTimeDelay(5);

}

}

### 驱动移植can\_stm32.c

#### 实现接收处理void \* can\_rx\_handle(void \* parameters)

void \* can\_rx\_handle(void \* parameters)

{

static uint8\_t s\_rx\_idlenum =0;

uint8\_t i;

can\_frame\_t frame;

driver\_can\_data\_t tmp\_can\_recvdata;

if((int8\_t)0 == driver\_can\_recv(&tmp\_can\_recvdata, (uint32\_t)1000))

{

frame.id = tmp\_can\_recvdata.id;

if(tmp\_can\_recvdata.len > (uint8\_t)8)

{

frame.dlc = (uint8\_t)8;

}

else

{

frame.dlc = tmp\_can\_recvdata.len;

}

for(i=0;i<frame.dlc;i++)

{

frame.data[i] = tmp\_can\_recvdata.buff[i];

}

/\* Call RX callback \*/

csp\_can\_process\_frame(&frame);

s\_rx\_idlenum=0;

}

else

{

s\_rx\_idlenum++;

if(s\_rx\_idlenum>=10)

{

s\_rx\_idlenum=0;

csp\_can\_pbuf\_cleanup();

}

}

return 0;

}

#### 发送一帧int can\_send(can\_id\_t id, uint8\_t data[], uint8\_t dlc)

#### 初始化int can\_init(uint32\_t id, uint32\_t mask, struct csp\_can\_config \*conf)

初始化主要是ID过滤和波特率

### 系统资源移植src/arch/freertos include/arch/freertos文件夹下

实现对消息队列，信号量互斥量、内存管理、时间、任务创建、系统相关(复位，获取剩余堆大小，关机、设置终端颜色等接口)

### Clock.h

实现该头文件要求的时基函数

## 问**题**

**内存释放混乱，一处分配到处释放。违背了谁分配谁释放，分配释放成对集中的原则。**

### 在底层发送csp\_can\_tx时释放csp\_buffer\_free

csp\_if\_can.c文件中csp\_can\_tx函数最后调用csp\_buffer\_free(packet);

对传入的packet释放了,

如果调用者先是packet =csp\_buffer\_free获取内存然后csp\_can\_tx发送后packet就失效了。

内存应该是谁分配谁负责释放，成对出现，尽量分配释放集中在一起。由csp\_can\_tx释放会导致混乱。

比如例程kiss.c 服务端程序CSP\_DEFINE\_TASK(task\_server)

就有问题

response是动态分配的，调用csp\_send(conn, response, 1000);后就会释放response，

下次再调用csp\_send(conn, response, 1000);时 respomse的内容就被改写了(如果该块重新被分配并且使用)

### 重复释放或者某些条件未释放

csp\_io.c文件中,函数csp\_transaction\_persistent

发送数据时临时分配了内存进行存储

int size = (inlen > outlen) ? inlen : outlen;

csp\_packet\_t \* packet = csp\_buffer\_get(size);

if (packet == NULL)

return 0;

/\* Copy the request \*/

if (outlen > 0 && outbuf != NULL)

memcpy(packet->data, outbuf, outlen);

packet->length = outlen;

if (!csp\_send(conn, packet, timeout)) {

csp\_buffer\_free(packet);

return 0;

}

如果按照原来的方式csp\_can\_tx中释放，则

if (!csp\_send(conn, packet, timeout)) {

csp\_buffer\_free(packet);

return 0;

}

失败是会释放两次。

如果不在csp\_can\_tx中释放释放则 成功时没释放。

有if没有else处理。

修改为 不在csp\_can\_tx中释放。

if (!csp\_send(conn, packet, timeout)) {

csp\_buffer\_free(packet);

return 0;

}

csp\_buffer\_free(packet); /\*成功发送完必须释放\*/

### 变量重复

csp\_service\_handler.c中csp\_service\_handler传入参数packet，

在后面又packet =csp\_buffer\_get(CSP\_RPS\_MTU);指向新分配的空间。

新用一个ps\_packet去指向分配空间。

### 释放混乱

同样是csp\_service\_handler.c中csp\_service\_handler，csp\_buffer\_free在各处释放，很混乱。

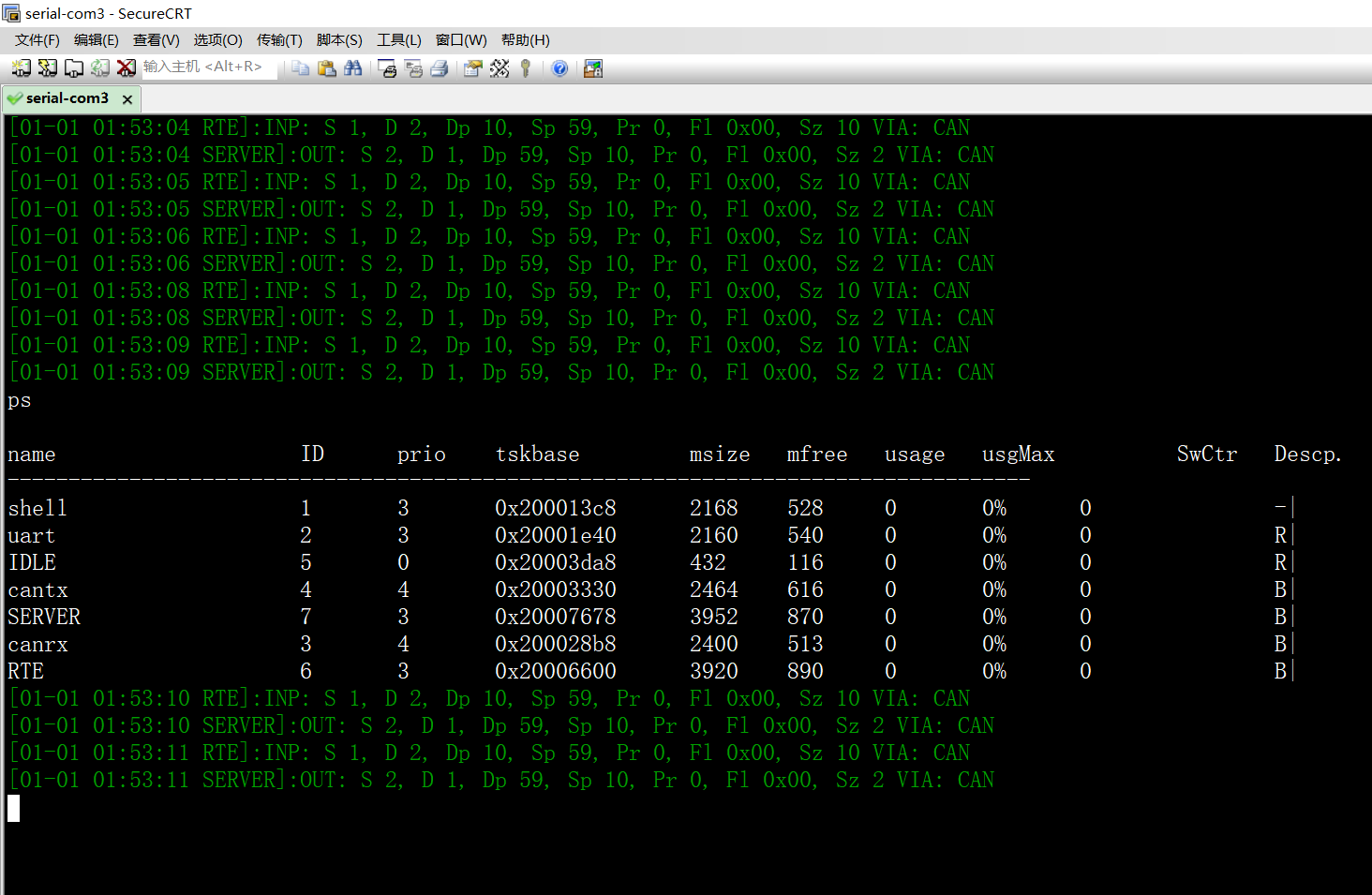
应该由调用者去释放。

### 代码不是很规范，注释较少。一些逻辑分支未处理。

### 直接内存操作，指针强制转换等操作较多。

## 测试环境

两个带CAN的stm32开发板 一个做客户端一个做服务端。客户端不断给服务端发送数据，服务端返回数据。



## 应用接口

csp\_io.c

## Todo

1. rdp，sfp协议需要详细分析
2. crypto加密部分需要了解