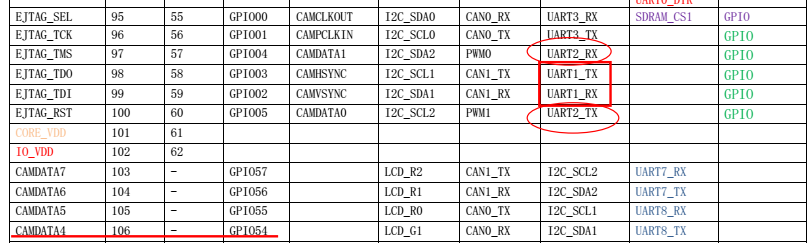
# 6. CAN总线驱动开发

## 6.1 智龙开发板硬件CAN接口

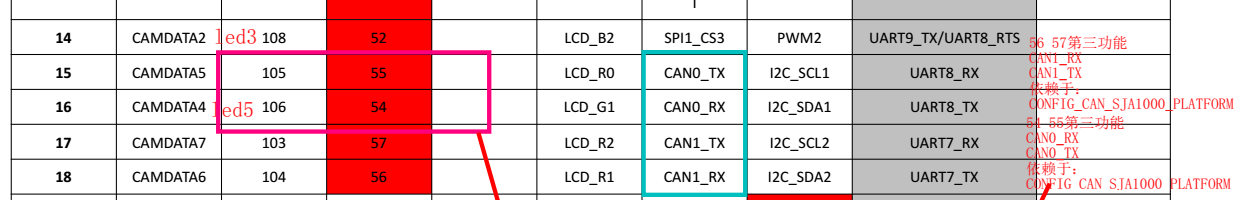
开发板的底板上电路图使用GPIO57，56—CAN1 GPIO55，54—CAN0。

**注意：扩展板丝印层上的CAN0\_H和CAN0\_L 错误，要对调。**

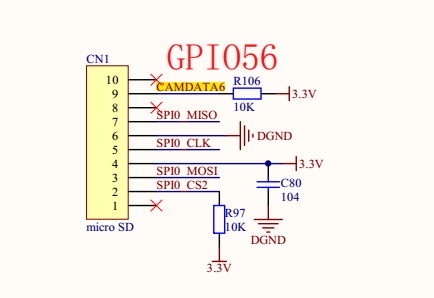




采用GPIO55、54，第三复用。复用成功后，LED5灯（orange）灭掉，不再动作。



GPIO56 用作SD卡探测，也不能用作CAN1。



内核配置中，启用CAN控制器。

[\*] Networking support --->

[\*] CAN bus subsystem support --->

--- CAN bus subsystem support

[\*] Raw CAN Protocol (raw access with CAN-ID filtering)

[\*] Broadcast Manager CAN Protocol (with content filtering)

CAN Device Drivers --->

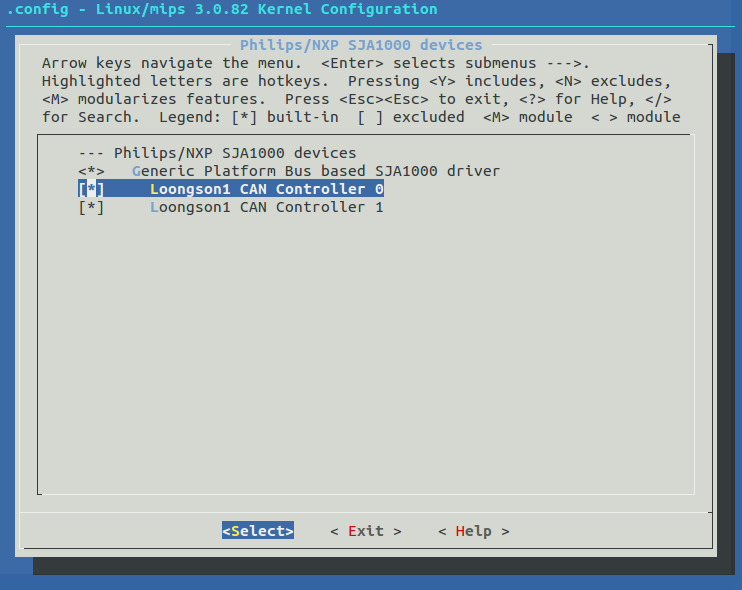
<\*> Philips/NXP SJA1000 devices --->

--- Philips/NXP SJA1000 devices --->

<\*> Generic Platform Bus based SJA1000 driver

[\*] Loong1CAN Controller 0

[\*] Loong1CAN Controller 1



在平台文件ls1c300a\_openloongson\_v2.0\_platform.c 中删除：

/\*54 用作CAN0 RX

{

.name = "led\_orange",

.gpio = 54,

.active\_low = 1,

.default\_trigger = "timer",

.default\_trigger = NULL,

.default\_state = LEDS\_GPIO\_DEFSTATE\_ON,

},\*/

在平台文件static void ls1x\_can\_setup(void) 函数中添加：

#ifdef CONFIG\_LS1X\_CAN0

/\* 设置复用关系 can0 gpio54/55 \*/

\_\_raw\_writel(\_\_raw\_readl(LS1X\_CBUS\_FIRST1) & (~0x00c00000), LS1X\_CBUS\_FIRST1);

\_\_raw\_writel(\_\_raw\_readl(LS1X\_CBUS\_SECOND1) & (~0x00c00000), LS1X\_CBUS\_SECOND1);

\_\_raw\_writel(\_\_raw\_readl(LS1X\_CBUS\_THIRD1) | 0x00c00000, LS1X\_CBUS\_THIRD1);

\_\_raw\_writel(\_\_raw\_readl(LS1X\_CBUS\_FOURTHT1) & (~0x00c00000), LS1X\_CBUS\_FOURTHT1);

/\* 使能can0控制器 \*/

\_\_raw\_writel(\_\_raw\_readl(LS1X\_MUX\_CTRL0) & (~CAN0\_SHUT), LS1X\_MUX\_CTRL0);

#endif //#ifdef CONFIG\_LS1X\_CAN0

#ifdef CONFIG\_LS1X\_CAN1

/\* 设置复用关系 can1 gpio56/57 \*/

\_\_raw\_writel(readl(LS1X\_CBUS\_FIRST1) &(~( 1 << (57 - 32))) ,LS1X\_CBUS\_FIRST1);

\_\_raw\_writel(readl(LS1X\_CBUS\_FIRST1) &( ~(1 << (56 - 32))) ,LS1X\_CBUS\_FIRST1);

\_\_raw\_writel(readl(LS1X\_CBUS\_SECOND1) &(~( 1 << (57 - 32))) ,LS1X\_CBUS\_SECOND1);

\_\_raw\_writel(readl(LS1X\_CBUS\_SECOND1) &( ~(1 << (56 - 32))) ,LS1X\_CBUS\_SECOND1);

\_\_raw\_writel(readl(LS1X\_CBUS\_FOURTHT1) &(~( 1 << (57 - 32))) ,LS1X\_CBUS\_FOURTHT1);

\_\_raw\_writel(readl(LS1X\_CBUS\_FOURTHT1) &( ~(1 << (56 - 32))) ,LS1X\_CBUS\_FOURTHT1);

\_\_raw\_writel(readl(LS1X\_CBUS\_THIRD1) |( 1 << (57 - 32)) ,LS1X\_CBUS\_THIRD1);

\_\_raw\_writel(readl(LS1X\_CBUS\_THIRD1) |( 1 << (56 - 32)) ,LS1X\_CBUS\_THIRD1); \*/

/\* 使能can1控制器 \*/

\_\_raw\_writel(\_\_raw\_readl(LS1X\_MUX\_CTRL0) & (~CAN1\_SHUT), LS1X\_MUX\_CTRL0);

#endif //#ifdef CONFIG\_LS1X\_CAN1

## 6.2 SocketCAN

原文地址：这篇文章主要针对can协议簇（aka socket can)

http://lxr.linux.no/linux+v2.6.34/Documentation/networking/can.txt

但是这篇文档没有涉及广播管理协议套接字 (SOCK\_DGRAM) 的内容。

另外一篇比较好的Socket CAN的英文文档是（详细介绍了广播管理协议套接字）：

Low Level CAN Framework Application Programmers Interface

http://www.brownhat.org/docs/socketcan/llcf-api.html#SECTION00051000000000000000

### 6.2.1 概述--什么是Socket CAN？

socketcan子系统是在Linux下CAN协议(Controller Area Network)实现的一种实现方法。 CAN是一种在世界范围内广泛用于自动控制、嵌入式设备和汽车领域的网络技术。Linux下最早使用CAN的方法是基于字符设备来实现的，与之不同的是Socket CAN使用伯克利的socket接口和linux网络协议栈，这种方法使得can设备驱动可以通过网络接口来调用。Socket CAN的接口被设计的尽量接近TCP/IP的协议，让那些熟悉网络编程的程序员能够比较容易的学习和使用。

### 6.2.2 动机--为什么使用socket API接口？

在Socket CAN之前Linux中已经有了一些CAN的实现方法，那为什么还要启动Socket CAN这个项目呢？大多数已经存在的实现方法仅仅作为某个具体硬件的设备驱动，它们往往基于字符设备并且提供的功能很少。那些方案通常是由一个针对具体硬件的设备驱动提供的字符设备接口来实现原始can帧的发送和接收，并且直接和控制器硬件打交道。帧队列和ISO-TP这样的高层协议必须在用户空间来实现。就像串口设备接口一样，大多数基于字符设备的实现在同一时刻仅仅支持一个进程的访问。如果更换了CAN控制器，那么同时也要更换另一个设备驱动，并且需要大多数应用程序重新调整以适应新驱动的API。

Socket CAN被设计用来克服以上种种不足。这种新的协议族实现了用户空间的socket接口，它构建于Linux网络层之上，因此可以直接使用已有的队列功能。CAN控制器的设备驱动将自己作为一个网络设备注册进Linux的网络层，CAN控制器收到的CAN帧可以传输给高层的网络协议和CAN协议族，反之，发送的帧也会通过高层给CAN控制器。传输协议模块可以使用协议族提供的接口注册自己，所以可以动态的加载和卸载多个传输协议。事实上，CAN核心模块不提供任何协议，也不能在没有加载其它协议的情况下单独使用。同一时间可以在相同或者不同的协议上打开多个套接字，可以在相同或者不同的CAN ID上同时监听和发送(listen/send)。几个同时监听具有相同ID帧的套接字可以在匹配的帧到来后接收到相同的内容。如果一个应用程序希望使用一个特殊的协议（比如ISO-TP）进行通信，只要在打开套接字的时候选择那个协议就可以了，接下来就可以读取和写入应用数据流了，根本无需关心CAN-ID和帧的结构等信息。

使用字符设备也可以让用户空间获得类似的便利，但是这中解决方案在技术上不够优雅，原因如下：

\*复杂的操作。使用Socket CAN，只需要向socket(2)函数传递协议参数并使用bind(2)选择CAN接口和CAN ID，基于字符设备实现这样的功能却要使用ioctl(2)来完成所有的操作。

\*无法代码复用。字符设备无法使用Linux的网络队列代码，所以必须为CAN 网络重写这部分功能。

\*缺乏抽象性。在大多数已经实现的字符设备方案中，硬件相关的CAN控制器设备驱动直接提供应用程序需要的字符设备。在Unix系统中，无论对于字符设备还是块设备，这都是不常见的。比如，你不会为某个串口、电脑上的某个声卡、访问磁带和硬盘的SCSI/IDE控制器直接创建一个字符设备。相反，你会将向应用程序提供统一字符设备/块设备接口的功能交给一个抽象层来做，你自己仅仅提供硬件相关的设备驱动接口。这种抽象是通过子系统来完成的，比如tty层子系统、声卡子系统和SCSI/IDE子系统。

实现CAN设备驱动最简单的办法就是直接提供一个字符设备而不使用（或不完全使用）抽象层，大多数已经存在的驱动也是这么做的。但是正确的方法却要增加抽象层以支持各种功能，如注册一个特定的CAN-ID，支持多次打开的操作和这些操作之间的CAN帧复用，支持CAN帧复杂的队列功能，还要提供注册设备驱动的API。然而使用Linux内核提供的网络框架将会大大简化，这就是Socket CAN要做的。

在Linux中实现CAN功能最自然和合适的方式就是使用内核的网络框架。

### 6.2.3 Socket CAN详解

就像第二章所讲的那样，使用Socket CAN的主要目的就是为用户空间的应用程序提供基于Linux网络层的套接字接口。与广为人知的TCP/IP协议以及以太网不同，CAN总线没有类似以太网的MAC层地址，只能用于广播。CAN ID仅仅用来进行总线的仲裁。因此CAN ID在总线上必须是唯一的。当设计一个CAN-ECU(Electronic Control Unit 电子控制单元）网络的时候，CAN-ID可以映射到具体的ECU。因此CAN-ID可以当作发送源的地址来使用。

#### 6.2.3.1 接收队列

允许多个应用程序同时访问网络导致了新的问题出现，那就是不同的应用程序可能会在同一个CAN网络接口上对具有相同CAN-ID的帧感兴趣。Socket CAN的核心部分实现了Socket CAN的协议族，通过高效的接收队列解决了这个问题。比如一个用户空间的程序打开了一个原始CAN套接字，原始协议模块将向CAN套接字的核心模块申请用户空间需要的一系列CAN-ID。Socket CAN的核心向CAN协议模块提供预约和解约CAN-ID的接口--can\_rx\_(un)register()，无论这个CAN-ID是针对一个具体的CAN接口还是所有已知的CAN接口（参考第5章）。

为了优化CPU的运行效率，每个设备都对应一个接收队列，这样比较容易实现各种报文过滤规则。

#### 6.2.3.2 发送帧的本地回环

在其它种类的网络中，在相同或者不同网络节点上的应用程序都可以相互交换数据。

\_\_\_ \_\_\_ \_\_\_ \_\_\_\_\_\_\_ \_\_\_

| \_ | | \_ | | \_ | | \_ \_ | | \_ |

||A|| ||B|| ||C|| ||A| |B|| ||C||

|\_\_\_| |\_\_\_| |\_\_\_| |\_\_\_\_\_\_\_| |\_\_\_|

| | | | |

-----------------(1)- CAN bus -(2)---------------

请看上图的两个例子。为了保证应用程序A在两个例子中能够接收到同样的例子（例2中A和B在同一个CAN设备上），发送出去的CAN帧需要能够在本地回环。

Linux下的网络设备仅仅处理物理媒介上帧的发送和接受。总线仲裁机制下，高优先级的帧会将低优先级的帧延后。为了正确反映实际的通信活动，回环必须在正确传输成功之后进行。如果CAN网络的硬件不支持回环功能，一种低效的方案是使用Socket CAN核心部分来实现软件回环。具体的情况请参考6.2小节。

CAN网络的回环功能是默认开启的。由于RT-SocketCAN 的特殊需求，每个套接字的回环功能可以被独立关闭。CAN原始套接字的控制选项请参考4.1小节。

\*当你在同一个节点上运行CAN分析命令“candump”或者“cansniffer”的时候就会发现回环功能真的很有用。

#### 6.2.3.3 网络安全相关

CAN网络是一种现场总线，仅仅支持没有路由和安全控制的广播传输。大部分应用程序都需要要直接处理原始CAN帧，所以和其它类型的网络一样，CAN网络对所有的用户（而不仅仅是root用户）访问没有任何限制。由于当前CAN\_RAW和CAN\_BCM的实现仅仅支持对CAN接口的读写，所以允许所有的用户访问CAN并不影响其它类型网络的安全性。为了使能非root用户对CAN\_RAW和CAN\_BCM协议套接字的访问，必须在编译内核的时候选上Kconfig的CAN\_RAW\_USER/CAN\_BCM\_USER选项。

#### 6.2.3.4 网络故障监测

使用CAN总线的时候，可能会遇到物理和mac（media access control）层的问题。为了方便用户分析物理收发器的硬件错误、总线仲裁错误和不同的ECU( Electrical Conversion Unit,电气转换装置)引起的错误，对于底层（物理和mac层）的监测和记录是至关重要的。拥有精确时间戳的错误监测对于诊断错误是非常重要的。基于以上原因，CAN接口的驱动可以可以选择性的产生所谓的错误帧，这些帧以和其它的CAN帧一样的方式传递给用户程序。当一个物理层或者MAC层的错误被(CAN控制器)检测到之后，驱动创建一个相应的错误帧。错误帧可以被应用程序通过CAN的过滤机制请求得到。过滤机制允许选择需要的错误帧的类型。默认情况下，接收错误帧的功能是禁止的。

CAN错误帧的详细格式定义在linux头文件中：include/linux/can/error.h。

### 6.2.4 如何使用Socket CAN

就像TCP/IP协议一样，在使用CAN网络之前你首先需要打开一个套接字。CAN的套接字使用到了一个新的协议族，所以在调用socket(2)这个系统函数的时候需要将PF\_CAN作为第一个参数。当前有两个CAN的协议可以选择，一个是原始套接字协议（ raw socket protocol），另一个是广播管理协议BCM（broadcast manager）。你可以这样来打开一个套接字：

s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);

或者

s = socket(PF\_CAN, SOCK\_DGRAM, CAN\_BCM);

在成功创建一个套接字之后，你通常需要使用bind(2)函数将套接字绑定在某个CAN接口上（这和TCP/IP使用不同的IP地址不同，参见第3章）。在绑定 (CAN\_RAW)或连接(CAN\_BCM)套接字之后，你可以在套接字上使用read(2)/write(2)，也可以使用send(2)/sendto(2)/sendmsg(2)和对应的recv\*操作。当然也会有CAN特有的套接字选项，下面将会说明。

基本的CAN帧结构体和套接字地址结构体定义在include/linux/can.h：

/\*

\* 扩展格式识别符由 29 位组成。其格式包含两个部分：11 位基本 ID、18 位扩展 ID。

\* Controller Area Network Identifier structure

\*

\* bit 0-28 : CAN识别符 (11/29 bit)

\* bit 29 : 错误帧标志 (0 = data frame, 1 = error frame)

\* bit 30 : 远程发送请求标志 (1 = rtr frame)

\* bit 31 :帧格式标志 (0 = standard 11 bit, 1 = extended 29 bit)

\*/

typedef \_\_u32 canid\_t;

struct can\_frame {

canid\_t can\_id; /\* 32 bit CAN\_ID + EFF/RTR/ERR flags \*/

\_\_u8 can\_dlc; /\* 数据长度: 0 .. 8 \*/

\_\_u8 data[8] \_\_attribute\_\_((aligned(8)));

};

结构体的有效数据在data[]数组中，它的字节对齐是64bit的，所以用户可以比较方便的在data[]中传输自己定义的结构体和共用体。CAN总线中没有默认的字节序。在CAN\_RAW套接字上调用read(2)，返回给用户空间的数据是一个struct can\_frame结构体。

就像PF\_PACKET套接字一样，sockaddr\_can结构体也有接口的索引，这个索引绑定了特定接口：

struct sockaddr\_can {

sa\_family\_t can\_family;

int can\_ifindex;

union {

/\* transport protocol class address info (e.g. ISOTP) \*/

struct { canid\_t rx\_id, tx\_id; } tp;

/\* reserved for future CAN protocols address information \*/

} can\_addr;

};

指定接口索引需要调用ioctl()（比如对于没有错误检查CAN\_RAW套接字）:

int s;

struct sockaddr\_can addr;

struct ifreq ifr;

s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);

strcpy(ifr.ifr\_name, "can0" );

ioctl(s, SIOCGIFINDEX, &ifr);

addr.can\_family = AF\_CAN;

addr.can\_ifindex = ifr.ifr\_ifindex;

bind(s, (struct sockaddr \*)&addr, sizeof(addr));

(..)

为了将套接字和所有的CAN接口绑定，接口索引必须是0。这样套接字便可以从所有使能的CAN接口接收CAN帧。recvfrom(2)可以指定从哪个接口接收。在一个已经和所有CAN接口绑定的套接字上，sendto(2)可以指定从哪个接口发送。

从一个CAN\_RAW套接字上读取CAN帧也就是读取struct can\_frame结构体:

struct can\_frame frame;

nbytes = read(s, &frame, sizeof(struct can\_frame));

if (nbytes < 0) {

perror("can raw socket read");

return 1;

}

/\* paranoid check ... \*/

if (nbytes < sizeof(struct can\_frame)) {

fprintf(stderr, "read: incomplete CAN frame\n");

return 1;

}

/\* do something with the received CAN frame \*/

写CAN帧也是类似的，需要用到write (2)函数：

nbytes = write(s, &frame, sizeof(struct can\_frame));

如果套接字跟所有的CAN接口都绑定了（addr.can\_index = 0），推荐使用recvfrom(2)获取数据源接口的信息：

struct sockaddr\_can addr;

struct ifreq ifr;

socklen\_t len = sizeof(addr);

struct can\_frame frame;

nbytes = recvfrom(s, &frame, sizeof(struct can\_frame),

0, (struct sockaddr\*)&addr, &len);

/\* get interface name of the received CAN frame \*/

ifr.ifr\_ifindex = addr.can\_ifindex;

ioctl(s, SIOCGIFNAME, &ifr);

printf("Received a CAN frame from interface %s", ifr.ifr\_name);

对于绑定了所有接口的套接字，向某个端口发送数据必须指定接口的详细信息：

strcpy(ifr.ifr\_name, "can0");

ioctl(s, SIOCGIFINDEX, &ifr);

addr.can\_ifindex = ifr.ifr\_ifindex;

addr.can\_family = AF\_CAN;

nbytes = sendto(s, &frame, sizeof(struct can\_frame),

0, (struct sockaddr\*)&addr, sizeof(addr));

#### 6.2.4.1 使用can\_filter的原始套接字 (RAW socket)

CAN\_RAW套接字的用法和CAN字符设备的用法是类似的。为了使用CAN套接字的新特性，在绑定原始套接字的时候将会默认开启以下特性：

- filter将会接收所有的数据

- 套接字仅仅接收有效的数据帧（=> no error frames）

- 发送帧的回环功能被开启（参见 3.2节）

- （回环模式下）套接字不接收它自己发送的帧

这些特性的设置可以在绑定之前和之后修改。为了使用CAN\_RAW套接字相关的选项，必须包含<linux/can/raw.h>。

#### 6.2.4.1.1 原始套接字选项 CAN\_RAW\_FILTER

CAN\_RAW套接字的接收可以使用CAN\_RAW\_FILTER套接字选项指定的多个过滤规则（过滤器）来过滤。

过滤规则（过滤器）的定义在 include/linux/can.h中:

struct can\_filter {

canid\_t can\_id;

canid\_t can\_mask;

};

过滤规则的匹配：

<received\_can\_id> & mask == can\_id & mask

/\*

#define CAN\_INV\_FILTER 0x20000000U /\* to be set in can\_filter.can\_id \*/

#define CAN\_ERR\_FLAG 0x20000000U /\* error frame \*/

\*/

这和大家所熟知的CAN控制器硬件过滤非常相似。可以使用 CAN\_INV\_FILTER这个宏将can\_filter结构体的成员can\_id中的比特位反转。和CAN控制器的硬件过滤形成鲜明对比的是，用户可以为每一个打开的套接字设置多个独立的过滤规则（过滤器）:

/\*

/\* valid bits in CAN ID for frame formats \*/

#define CAN\_SFF\_MASK 0x000007FFU /\* 标准帧格式 (SFF) \*/

#define CAN\_EFF\_MASK 0x1FFFFFFFU /\* 扩展帧格式 (EFF) \*/

#define CAN\_ERR\_MASK 0x1FFFFFFFU /\* 忽略EFF, RTR, ERR标志 \*/

\*/

struct can\_filter rfilter[2];

rfilter[0].can\_id = 0x123;

rfilter[0].can\_mask = CAN\_SFF\_MASK;

rfilter[1].can\_id = 0x200;

rfilter[1].can\_mask = 0x700;

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, &rfilter, sizeof(rfilter));

为了在指定的CAN\_RAW套接字上禁用接收过滤规则，可以这样：

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, NULL, 0);

在一些极端情况下不需要读取数据，可以把过滤规则清零（所有成员设为0），这样原始套接字就会忽略接收到的CAN帧。在这种仅仅发送数据（不读取）的应用中可以在内核中省略接收队列，以此减少CPU的负载（虽然只能减少一点点）。

#### 6.2.4.1.2 原始套接字选项 CAN\_RAW\_ERR\_FILTER

正如3.4节所说，CAN接口驱动可以选择性的产生错误帧，错误帧和正常帧以相同的方式传给应用程序。可能产生的错误被分为不同的种类，使用适当的错误掩码可以过滤它们。为了注册所有可能的错误情况，CAN\_ERR\_MASK（0x1FFFFFFFU）这个宏可以用来作为错误掩码。这个错误掩码定义在linux/can/error.h。

can\_err\_mask\_t err\_mask = ( CAN\_ERR\_TX\_TIMEOUT | CAN\_ERR\_BUSOFF );

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_ERR\_FILTER,

&err\_mask, sizeof(err\_mask));

#### 6.2.4.1.3 原始套接字选项 CAN\_RAW\_LOOPBACK

为了满足众多应用程序的需要，本地回环功能默认是开启的（详细情况参考3.2节）。但是在一些嵌入式应用场景中（比如只有一个用户在使用CAN总线），回环功能可以被关闭（各个套接字之间是独立的）：

int loopback = 0; /\* 0 = disabled, 1 = enabled (default) \*/

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_LOOPBACK, &loopback, sizeof(loopback));

#### 6.2.4.1.4 原始套接字选项 CAN\_RAW\_RECV\_OWN\_MSGS

在本地回环功能开启的情况下，所有的发送帧都会被回环到在相应CAN接口上注册了同样CAN-ID（和发送帧的相同）的套接字上。发送CAN帧的套接字被假设不想接收自己发送的CAN帧，因此在发送套接字上的回环功能默认是关闭的。可以在需要的时候改变这一默认行为：

int recv\_own\_msgs = 1; /\* 0 = disabled (default), 1 = enabled \*/

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_RECV\_OWN\_MSGS,

&recv\_own\_msgs, sizeof(recv\_own\_msgs));

#### 6.2.4.2 广播管理协议套接字 (SOCK\_DGRAM)

（没有内容）

#### 6.2.4.3 面向连接的传输协议 (SOCK\_SEQPACKET)

（没有内容）

#### 6.2.4.4 无连接的传输协议 (SOCK\_DGRAM)

（没有内容）

### 6.2.5 Socket CAN核心模块

CAN套接字的核心模块实现了PF\_CAN协议族。CAN协议模块在核心模块运行后加载。CAN核心模块为协议模块提供了申请需要的ID的接口（参考3.1小节）。

#### 6.2.5.1 can.ko模块的参数

- stats\_timer: 为了计算CAN套接字的统计信息（比如最近一秒的帧数和每秒最大的帧数），can.ko调用一个定时间隔为1秒的定时器，默认情况下这个定时器是开启的。这个定时器也可以在模块参数中传入stattimer=0来禁止。

- debug : (removed since SocketCAN SVN r546)

#### 6.2.5.2 procfs接口

就像3.1节描述的那样，CAN套接字核心借助于一些带有过滤规则的队列向CAN协议模块传递接收到的CAN帧。可以在procfs中查看这些接收队列的的过滤规则和匹配规则的次数。所有的条目都包了设备名和协议模块标识：

foo@bar:~$ cat /proc/net/can/rcvlist\_all

receive list 'rx\_all': (vcan3: no entry)

(vcan2: no entry)

(vcan1: no entry)

device can\_id can\_mask function userdata matches ident

vcan0 000 00000000 f88e6370 f6c6f400 0 raw

(any: no entry)

（yll补充：function是一个函数指针，userdata是一个void \*指针，所以这两个值打印出来没有太大意义）

在这个例子中，这个应用程序接收所有vcan0上传输的数据.

rcvlist\_all - 没有过滤规则的队列

rcvlist\_eff - 扩展帧（EFF）的队列

rcvlist\_err - 错误帧队列

rcvlist\_fil - 通过过滤规则的队列

rcvlist\_inv - 未通过过滤规则的队列

rcvlist\_sff - 标准帧的队列

在/proc/net/can中还有另外一些文件：

stats - CAN套接字核心的统计信息（接收/发送的帧数，匹配率等）

reset\_stats - 复位统计信息

version - 打印CAN套接字核心的版本和ABI的版本

#### 6.2.5.3 写一个自己的CAN协议模块

要在PF\_CAN中增加一个新的协议，必须在include/linux/can.h中为新的协议增加相应的定义。包含include/linux/can.h这个文件便可以使用增加的原型和定义。内核除了提供了注册CAN协议和CAN设备的通知列表的功能，也提供了在一个特定CAN接口上注册感兴趣的CAN帧或者发送CAN帧的功能。

can\_rx\_register - 在一个特定接口上注册希望接收到的CAN帧的信息 （yll:这个函数的定义在内核的net/can/af\_can.c中）

can\_rx\_unregister - 注销上面的申请

can\_send - 发送CAN帧（可以选择是否开启本地回环）

详细的信息请参考内核中的源码net/can/af\_can.c、net/can/raw.c、net/can/bcm.c。

### 6.2.6 CAN网络驱动

编写一个CAN网络设备驱动要比写一个CAN字符设备驱动要容易的多。和编写其它网络设备驱动类似，你只要处理以下事宜：

- TX :将套接字缓冲区的CAN帧发送到CAN控制器

- RX :从CAN控制器的CAN帧读取到套接字缓冲区

Documentation/networking/netdevices.txt中是网络设备驱动的例子。下面将会介绍CAN网络设备驱动的一些独有特性。

#### 6.2.6.1 常见设置

dev->type = ARPHRD\_CAN; /\* the netdevice hardware type \*/

dev->flags = IFF\_NOARP; /\* CAN has no arp \*/

dev->mtu = sizeof(struct can\_frame);

结构体can\_frame是PF\_CAN协议族套接字缓冲区的数组载体。

#### 6.2.6.2 发送帧的本地回环

如3.2小节所述，CAN网络设备驱动应该支持类似TTY设备回显功能的本地回环功能。如果驱动支持这个功能，则要设备IFF\_ECHO标志来防止PF\_CAN核心回显发送帧（又称回环）。

dev->flags = (IFF\_NOARP | IFF\_ECHO);

#### 6.2.6.3 CAN控制器的硬件过滤

为了减小一些嵌入式系统的中断负载，一些CAN控制器支持多个CAN-ID或者多个CAN-ID区间的过滤功能。硬件过滤功能在不同的控制器之间差异很大，并且不能同时满足多个用户的不同过滤需求。在单用户应用中使用控制器的硬件过滤或许还有意义，但是在一个多用户系统中驱动层的过滤将会影响所有用户。PF\_CAN核心内置的过滤规则集合允许对每个套接字独立的设置多个过滤规则。因此使用硬件过滤属于嵌入式系统中“手动调整”的范畴。从2002年开始笔者一直使用拥有四路SJA1000 CAN控制器的MPC603e @133MHz，总线负载虽然很高，但是到目前为止还没有什么问题。

#### 6.2.6.4 虚拟的CAN驱动 (vcan)

和网络回环设备一样，vcan提供一个虚拟的本地CAN接口。CAN中一个有效的地址包括：

- 一个有效的CAN标识符（CAN ID）

- 这个CAN标识符将要发往的总线（比如 can0）.

所以在一般的应用场景中往往需要多个vcan接口。

vcan接口允许在没有控制器硬件的情况下进行发送和接收。vcan网络设备的命名一般采用‘vcanX’,比如can1 vcan2等。当编译为单独的模块的时候，vcan驱动的模块名为vcan.ko。

vcan驱动从linux2.6.24开始支持netlink接口，使得创建vcan网络设备变的可能。可以使用ip(8)命令工具管理vcan网络设备的创建和移除：

- 创建一个vcan网络接口：

$ ip link add type vcan

- 使用给定的名字 'vcan42'创建 一个vcan网络接口：

$ ip link add dev vcan42 type vcan

- 移除vcan网络接口'vcan42':

$ ip link del vcan42

#### 6.2.6.5 CAN网络设备驱动接口

CAN网络设备驱动提供了进行安装、配置和监控CAN网络设备的接口。可以使用IPROUTE2工具集中的“ip”命令通过netlink接口配置CAN设备，比如设置波特率。本章剩余的部分将会简介如何使用这一工具。另外这些接口使用一些通用的数据结构并且提供了一系列常用的功能，这些功能都是CAN网络设备驱动需要用到的。请参考SJA1000或者MSCAN的驱动去了解如何使用它们。模块的名字是can-dev.ko。

#### 6.2.6.5.1 Netlink接口--设置/获取设备属性

CAN设备必须使用netlink来配置。在"include/linux/can/netlink.h"有对netlink消息类型的定义和简短描述。IPROUTE2工具集中的“ip”命令可以使用CAN的netlink支持，下面是使用示例：

- 设置CAN设备属性：

$ ip link set can0 type can help

Usage: ip link set DEVICE type can [ bitrate BITRATE [ sample-point SAMPLE-POINT] ] |

[ tq TQ prop-seg PROP\_SEG phase-seg1 PHASE-SEG1 phase-seg2 PHASE-SEG2 [ sjw SJW ] ]

[ loopback { on | off } ]

[ listen-only { on | off } ]

[ triple-sampling { on | off } ]

[ restart-ms TIME-MS ]

[ restart ]

Where: BITRATE := { 1..1000000 } SAMPLE-POINT := { 0.000..0.999 }

TQ := { NUMBER }

PROP-SEG := { 1..8 }

PHASE-SEG1 := { 1..8 }

PHASE-SEG2 := { 1..8 }

SJW := { 1..4 }

RESTART-MS := { 0 | NUMBER }

- 显示CAN设备的详情和统计信息：

$ ip -details -statistics link show can0

2: can0: <NOARP,UP,LOWER\_UP,ECHO> mtu 16 qdisc pfifo\_fast state UP qlen 10

link/can

can <TRIPLE-SAMPLING> state ERROR-ACTIVE restart-ms 100

bitrate 125000 sample\_point 0.875

tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1

sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1

clock 8000000

re-started bus-errors arbit-lost error-warn error-pass bus-off

41 17457 0 41 42 41

RX: bytes packets errors dropped overrun mcast

140859 17608 17457 0 0 0

TX: bytes packets errors dropped carrier collsns

861 112 0 41 0 0

下面是上面一些名词的解释：

"<TRIPLE-SAMPLING>"

表示选中的CAN控制器的模式：LOOPBACK, LISTEN-ONLY, or TRIPLE-SAMPLING。

"state ERROR-ACTIVE"

CAN控制器的当前状态："ERROR-ACTIVE", "ERROR-WARNING", "ERROR-PASSIVE", "BUS-OFF" or "STOPPED"

"restart-ms 100"

自动重启的延时时间。如果设为非零值， 在总线关闭的情况下，在设定的数量毫秒后CAN控制器被自动触发。这个功能默认是关闭的。

"bitrate 125000 sample\_point 0.875"

使用bits/sec作为单位显示位时间并显示0.000～0.999的采样点位置。如果内核中使能了统计位时间的功能(CONFIG\_CAN\_CALC\_BITTIMING=y)，位时间可以使用"bitrate"参数来设置。可选的"sample-point"也是可以配置的。默认使用0.000这个推荐值。

"tq 125 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1"

以ns为单位显示时间份额（tq-time quanta）、传播段（prop-seg ： propagation segment）、相位缓冲段1和2（phase-seg：phase buffer），以tq为单位显示同步跳转宽度（sjw：synchronisation jump width）。这些变量允许定义与硬件无关的位时序，这也是Bosch CAN 2.0 spec所推荐的（参考第八章http://www.semiconductors.bosch.de/pdf/can2spec.pdf）。

"sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1 clock 8000000"

显示CAN控制器的比特时序常量，这里的例子以sja1000为例。时间段（tseg -time segment）1和2的最小和最大值，以tq为单位的同步跳转宽度，比特速率的预分频器（brp--pre-scaler）和CAN系统时钟（以HZ为单位）。这些常量可以被用户空间的比特时序统计算法所使用。

"re-started bus-errors arbit-lost error-warn error-pass bus-off"

显示重启的次数、总线和仲裁丢失错误，错误主动（error-warning）、错误被动（error-passive）、和总线关闭的状态变化。接收的过载错误在 统计信息的"overrun"域下面列出。

#### 6.2.6.5.2 设置CAN的比特\_时序

CAN比特时序参数可以使用硬件无关的定义方法。这些参数是： "tq", "prop\_seg", "phase\_seg1", "phase\_seg2" 和 "sjw":

$ ip link set canX type can tq 125 prop-seg 6 \

phase-seg1 7 phase-seg2 2 sjw 1

在内核选项CONFIG\_CAN\_CALC\_BITTIMING被使能的情况下，如果比特率（波特率）参数 "bitrate"被设置了，CAN的这些参数将会生效：

$ ip link set canX type can bitrate 125000

请注意，这条命令在大部分使用标准波特率的CAN控制器上工作良好，但是使用一些少见的波特率值（如115000）和时钟频率值将会失败。禁用内核的CONFIG\_CAN\_CALC\_BITTIMING选项可以节省一些内存空间并且允许用户空间的命令工具完全的控制比特时序参数。使用CAN控制器的比特时序常量就可以达到这个目的（用户空间控制比特时序）。下面的命令将会列出这些变量：

$ ip -details link show can0

...

sja1000: clock 8000000 tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1

#### 6.2.6.5.3 启动和停止CAN网络设备

一个CAN网络设备可以使用"ifconfig canX up/down" 或者 "ip link set canX up/down"来开启和关闭。为了避免错误的默认值，必须在启动CAN设备之前设置它的比特时序参数：

$ ip link set canX up type can bitrate 125000

如果总线上出现太多的错误设备可能进入总线关闭状态（也就是从总线脱离）。进入总线关闭状态之后设备不会再发送和接收信息。给"restart-ms"设置一个非零值可以开启总线关闭自动恢复的功能（也就是进入总线关闭状态后重新开启），下面是一个示例：

$ ip link set canX type can restart-ms 100

应用程序可以通过监测CAN的错误帧意识到已经进入总线关闭状态，并且可以使用以下命令重启：

$ ip link set canX type can restart

注意，重启也会生成一个CAN错误帧（参见3.4节）。

#### 6.2.6.6 支持Socket CAN的硬件

"drivers/net/can"中的“Kconfig”文件中可以查看到所有支持的硬件列表。在CAN套接字项目网站上（参见第7章）有更多驱动何以获得，当然也有对早些时期版本内核的支持 。

### 6.2.7 学习Socket CAN的相关资源

你可在BerliOS OSS项目网站的CAN套接字页面中发现更多资源，比如用户空间工具、对旧版内核的支持、更多的驱动、邮件列表等：

<http://developer.berlios.de/projects/socketcan>

如果你有任何问题或者发现了BUG，不要迟疑，立马发送邮件到CAN套接字的用户邮件列表。但是在发送之前请首先搜索邮件记录中是否已经有了相同的问题。

### 6.2.8. 贡献者名单

=========

Oliver Hartkopp (PF\_CAN core, filters, drivers, bcm, SJA1000 driver) Urs Thuermann (PF\_CAN core, kernel integration, socket interfaces, raw, vcan) Jan Kizka (RT-SocketCAN core, Socket-API reconciliation)

Wolfgang Grandegger (RT-SocketCAN core & drivers, Raw Socket-API reviews, CAN device driver interface, MSCAN driver) Robert Schwebel (design reviews, PTXdist integration)

Marc Kleine-Budde (design reviews, Kernel 2.6 cleanups, drivers) Benedikt Spranger (reviews)

Thomas Gleixner (LKML reviews, coding style, posting hints)

Andrey Volkov (kernel subtree structure, ioctls, MSCAN driver) Matthias Brukner (first SJA1000 CAN netdevice implementation Q2/2003) Klaus Hitschler (PEAK driver integration)

Uwe Koppe (CAN netdevices with PF\_PACKET approach)

Michael Schulze (driver layer loopback requirement, RT CAN drivers review) Pavel Pisa (Bit-timing calculation)

Sascha Hauer (SJA1000 platform driver)

Sebastian Haas (SJA1000 EMS PCI driver)

Markus Plessing (SJA1000 EMS PCI driver)

Per Dalen (SJA1000 Kvaser PCI driver)

Sam Ravnborg (reviews, coding style, kbuild help)

============================================

作者：yuanlulu

http://blog.csdn.net/yuanlulu

版权没有，但是转载请保留此段声明

## 6.3 测试工具

测试can需要ip,can-utils和libsocketcan库。通过ip工具配置can,如速率,启用和禁用can等。

## 6.4 Socket CAN在智龙上的测试-使用工具iproute2

ifocnfig -a 能看到can节点

[root@Loongson:/ can\_tools]#ifconfig -a

can0 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00

NOARP MTU:16 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:10

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:6

### 6.4.1 下载编译

1）下载源码

地址： <http://pkgs.fedoraproject.org/repo/pkgs/iproute/>

这里下载的是iproute2-3.1.0.tar.bz2。

或者 在虚拟机上运行以下命令

git clone git://git.kernel.org/pub/scm/linux/kernel/git/shemminger/iproute2.git

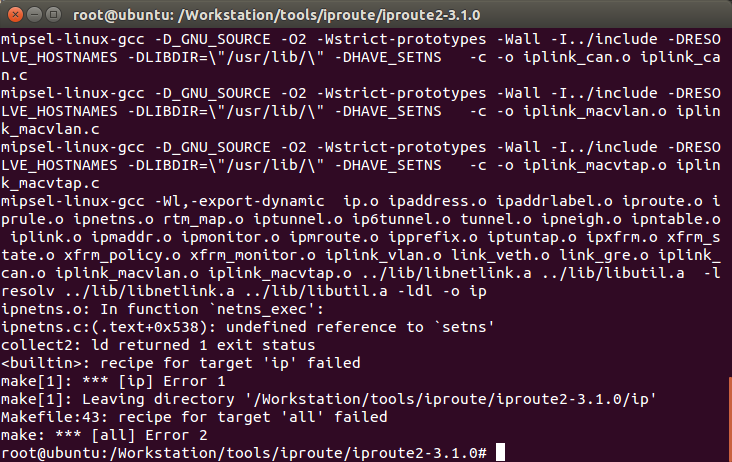
2）Makefile中修改交叉编译工具链

CC = mipsel-linux-gcc

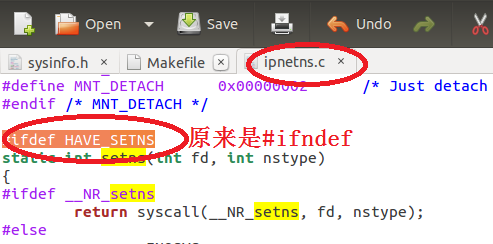
HOSTCC = mipsel-linux-gcc

SUBDIRS=lib ip

make编译后有错如下图：

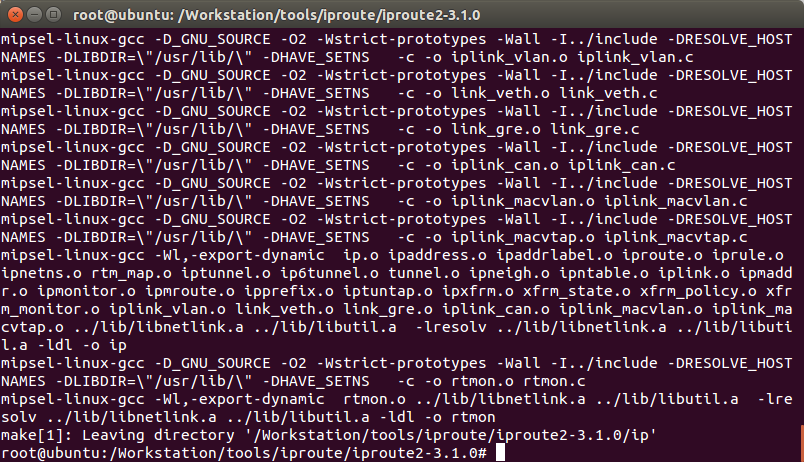


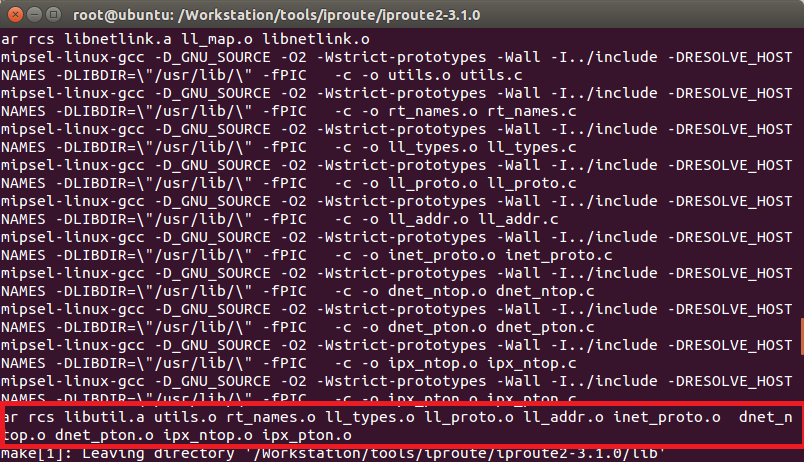
根据需要修改内核文件ipnetns.c文件，屏蔽调用setns的函数。

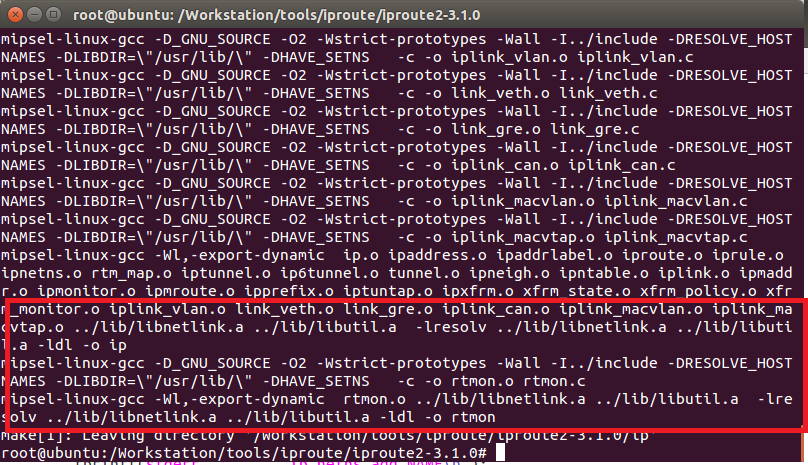


3）Make

再次编译后成功。生成IP目录。







4)拷备ip 文件到开发板

将ip工具copy到开发板的文件系统下，这里拷备到/can\_tools/下。

[root@Loongson:/can\_tools]#tftp -r ip -g 193.169.2.215

ip 100% |\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*| 290k 0:00:00 ETA

5)在开发板上应用IP link的命令查看设备

[root@Loongson:/can\_tools]#./ip link

-/bin/sh: ./can\_tools: Permission denied

[root@Loongson:/can\_tools]#chmod u+x ip

[root@Loongson:/can\_tools]#./ip link

1: lo: <LOOPBACK,UP,LOWER\_UP> mtu 16436 qdisc noqueue state UNKNOWN

link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00

2: can0: <NOARP,ECHO> mtu 16 qdisc noop state DOWN qlen 10

link/can

3: eth0: <BROADCAST,MULTICAST,UP,LOWER\_UP> mtu 1500 qdisc pfifo\_fast state UNKNOWN qlen 1000

link/ether aa:37:80:e3:d9:f9 brd ff:ff:ff:ff:ff:ff

4: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN qlen 1000

link/ether 00:0b:81:96:76:24 brd ff:ff:ff:ff:ff:ff

### 6.4.2 配置运行命令

配置, can接口需要在down状态下才能配置

[root@Loongson:/can\_tools]#./ip link set can0 down

[root@Loongson:/can\_tools]#./ip link set can0 up type can bitrate 250000

sja1000\_platform sja1000\_platform.0: setting BTR0=0x23 BTR1=0x04

[root@Loongson:/can\_tools]#./ip -details link show can0

2: can0: <NOARP,UP,LOWER\_UP,ECHO> mtu 16 qdisc pfifo\_fast state UNKNOWN qlen 10

link/can

can state ERROR-ACTIVE (berr-counter tx 0 rx 0) restart-ms 0

bitrate 250000 sample-point 0.857

tq 571 prop-seg 2 phase-seg1 3 phase-seg2 1 sjw 1

sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1

clock 63000000

down下的状态

[root@Loongson:/can\_tools]#./ip link set can0 down

[root@Loongson:/can\_tools]#./ip -details link show can0

2: can0: <NOARP,ECHO> mtu 16 qdisc pfifo\_fast state DOWN qlen 10

link/can

can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0

bitrate 250000 sample-point 0.857

tq 571 prop-seg 2 phase-seg1 3 phase-seg2 1 sjw 1

sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1

clock 63000000

启动

[root@Loongson:/can\_tools]# ifconfig can0 up

[root@Loongson:/can\_tools]#ifconfig

can0 Link encap:UNSPEC HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00

UP RUNNING NOARP MTU:16 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:10

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:6

loopback状态loopback状态，使用loopback模式可测试本机的can驱动是否正常工作.

[root@Loongson:/can\_tools]#./ip link set can0 down

[root@Loongson:/can\_tools]#./ip link set can0 type can bitrate 25000 loopback on

RTNETLINK answers: Operation not supported

[root@Loongson:/can\_tools]# ./ip -details link show can0

2: can0: <NOARP,ECHO> mtu 16 qdisc pfifo\_fast state DOWN qlen 10

link/can

can state STOPPED (berr-counter tx 0 rx 0) restart-ms 0

bitrate 250000 sample-point 0.857

tq 571 prop-seg 2 phase-seg1 3 phase-seg2 1 sjw 1

sja1000: tseg1 1..16 tseg2 1..8 sjw 1..4 brp 1..64 brp-inc 1

clock 63000000

回环模式不支持。

## 6.5 Socket CAN在智龙上的测试-使用工具canutils

can-utils和libsocketcan可以使用buildroot编译。

Target packages > Libraries > Networking

[\*] libsocketcan

Target packages > Networking applications

[\*] can-utils

由本系统采用的busybox, 这里重新下载canutils-4.0.6.tar.bz2编译。

### 6.5.1 安装libsocketcan

1）下载libsocketcan

canutils编译需要libsocketcan库的支持，需要下载libsocketcan。

下载地址：http://www.pengutronix.de/software/libsocketcan/download/ 笔者下载的是libsocketcan 0.0.10

2）配置编译工具链

--host是指定交叉工具链

./configure --host=mipsel-linux

3)编译libsocketcan - make

root@ubuntu:/Workstation/tools/SocketCAN/libsocketcan-0.0.10# make

Making all in include

make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'

make all-am

make[2]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'

make[2]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'

make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include'

Making all in config

make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/config'

make[1]: Nothing to be done for 'all'.

make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/config'

Making all in src

make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src'

CC libsocketcan.lo

CCLD libsocketcan.la

make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src'

make[1]: Entering directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10'

make[1]: Nothing to be done for 'all-am'.

make[1]: Leaving directory '/Workstation/tools/SocketCAN/libsocketcan-0.0.10'

在'/Workstation/tools/SocketCAN/libsocketcan-0.0.10/下生成了文件libsocketcan.lo、 CCLD libsocketcan.la 。

### 6.5.2 安装canutils

1）下载并配置canutils-4.0.6。

下载地址：<http://public.pengutronix.de/software/socket-can/canutils/> 。

这里下载canutils-4.0.6.tar.bz2。

解压canutils-4.0.6.tar.bz2，执行configure命令。（其中--host是指定交叉工具链，--prefix是指定库的生成位置(不用)，libsocketcan\_LIBS是指定canconfig需要链接的库，LDFLAGS是指定外部库的路径，CPPFLAGS是指定外部头文件的路径）

配置canutils

./configure --host=mipsel-linux libsocketcan\_LIBS=-lsocketcan LDFLAGS="L/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs " libsocketcan\_CFLAGS="-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include" CFLAGS="-I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include"

2）编译 make

# make

Making all in include

make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'

make all-am

make[2]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'

make[2]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'

make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/include'

Making all in config

make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/config'

make[1]: Nothing to be done for 'all'.

make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/config'

Making all in src

make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/src'

mipsel-linux-gcc -DHAVE\_CONFIG\_H -I. -I../include -I../include -I../include -DPF\_CAN=29 -DAF\_CAN=PF\_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT candump.o -MD -MP -MF .deps/candump.Tpo -c -o candump.o candump.c

mv -f .deps/candump.Tpo .deps/candump.Po

/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o candump candump.o

libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o candump candump.o

mipsel-linux-gcc -DHAVE\_CONFIG\_H -I. -I../include -I../include -I../include -DPF\_CAN=29 -DAF\_CAN=PF\_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT cansend.o -MD -MP -MF .deps/cansend.Tpo -c -o cansend.o cansend.c

mv -f .deps/cansend.Tpo .deps/cansend.Po

/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o cansend cansend.o

libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o cansend cansend.o

mipsel-linux-gcc -DHAVE\_CONFIG\_H -I. -I../include -I../include -I../include -DPF\_CAN=29 -DAF\_CAN=PF\_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT canecho.o -MD -MP -MF .deps/canecho.Tpo -c -o canecho.o canecho.c

mv -f .deps/canecho.Tpo .deps/canecho.Po

/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o canecho canecho.o

libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o canecho canecho.o

mipsel-linux-gcc -DHAVE\_CONFIG\_H -I. -I../include -I../include -I../include -DPF\_CAN=29 -DAF\_CAN=PF\_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT cansequence.o -MD -MP -MF .deps/cansequence.Tpo -c -o cansequence.o cansequence.c

mv -f .deps/cansequence.Tpo .deps/cansequence.Po

/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o cansequence cansequence.o

libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o cansequence cansequence.o

mipsel-linux-gcc -DHAVE\_CONFIG\_H -I. -I../include -I../include -I../include -DPF\_CAN=29 -DAF\_CAN=PF\_CAN -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -MT canconfig.o -MD -MP -MF .deps/canconfig.Tpo -c -o canconfig.o canconfig.c

mv -f .deps/canconfig.Tpo .deps/canconfig.Po

/bin/bash ../libtool --tag=CC --mode=link mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o canconfig canconfig.o -L/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs -lsocketcan

libtool: link: mipsel-linux-gcc -I/Workstation/tools/SocketCAN/libsocketcan-0.0.10/include -Wall -g -O2 -o .libs/canconfig canconfig.o -L/Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs /Workstation/tools/SocketCAN/libsocketcan-0.0.10/src/.libs/libsocketcan.so -Wl,-rpath -Wl,/usr/local/libsocketcan/lib

make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/src'

Making all in man

make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6/man'

make[1]: Nothing to be done for 'all'.

make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6/man'

make[1]: Entering directory '/Workstation/tools/SocketCAN/canutils-4.0.6'

make[1]: Nothing to be done for 'all-am'.

make[1]: Leaving directory '/Workstation/tools/SocketCAN/canutils-4.0.6'

在SRC 目录下生成candump cansend canecho cansequence 4个文件。

./lib下canconfig 1个文件。

3)将编译好的cansend 等5个文件复制到对应开发板/can\_tools下。

运行canconfig时出错：

[root@Loongson:/can\_tools]#./canconfig can0 up

./canconfig: error while loading shared libraries: libsocketcan.so.2: cannot open shared object file: No such file or directory

需将libsocketcan.so.2.2.1拷贝到开发板的/lib文件夹下，由于libsocketcan.so.2是链接文件，需要建立libsocketcan.so.2.2.1的快捷方式libsocketcan.so.2。

[root@Loongson:/lib]#tftp -r libsocketcan.so.2.2.1 -g 193.169.2.215

[root@Loongson:/lib]#ln -s libsocketcan.so.2.2.1 libsocketcan.so.2

附录：

虽然程序编译时是优先使用指定的库link,但是运行时，动态库还是从系统的位置找的.

因为想避免安装这个库，所需要将工程中动态库的地址手动加入到系统中。

主要有3种方法：

1. 用ln将需要的so文件链接到/usr/lib或者/lib这两个默认的目录下边

ln -s /where/you/install/lib/\*.so /usr/lib

sudo ldconfig

2.修改LD\_LIBRARY\_PATH

export LD\_LIBRARY\_PATH=/where/you/install/lib:$LD\_LIBRARY\_PATH

sudo ldconfig

3.修改/etc/ld.so.conf，然后刷新

vim /etc/ld.so.conf

add /where/you/install/lib

sudo ldconfig

### 6.5.3 使用canutils

1) 配置can0

[root@Loongson:/can\_tools]#./canconfig can0 bitrate 250000

sja1000\_platform sja1000\_platform.0: setting BTR0=0x23 BTR1=0x04

can0 bitrate: 250000, sample-point: 0.857

2） 开启 / 重启 / 关闭CAN总线

[root@Loongson:/can\_tools]#./canconfig can0 start

can0 state: ERROR-ACTIVE

[root@Loongson:/can\_tools]#./canconfig can0 restart

Device is not in BUS\_OFF, no use to restart

can0: failed to restart

[root@Loongson:/can\_tools]#./canconfig can0 stop

can0 state: STOPPED

[root@Loongson:/can\_tools]#

3) 查看CAN总线状态

[root@Loongson:/can\_tools]#./canecho can0

interface-in = can0, interface-out = can0, family = 29, type = 3, proto = 1

read: Network is down

4）发送信息

cansend canX --identifier=ID + 数据

[root@Loongson:/can\_tools]#./cansend can0 --identifier=0x123 0x11 0x22 0x33 0x44 0x55 0x66 0x77 0x88

interface = can0, family = 29, type = 3, proto = 1

write: Network is down

5)接收数据

candump canX

[root@Loongson:/can\_tools]#./candump can0

interface = can0, family = 29, type = 3, proto = 1

<0x001> [8] 00 01 02 03 04 05 06 07

<0x002> [8] 00 01 02 03 04 05 06 076)

使用滤波器接收ID匹配的数据

candump canX –-filter=ID:mask

[root@Loongson:/can\_tools]#./candump can0 --filter=0x123:0x7ff

id: 0x00000123 mask: 0x000007ff

interface = can0, family = 29, type = 3, proto = 1

7）总结

至此，使用Socket方式的CAN总线驱动设计的就介绍完了，用户可以使用Socket套接字的方式，参照canutils的源码设计自己的应用程序。

## 6.6 编写CAN的socket收发测试程序canapp

### 6.6.1 程序设计说明

下面具体介绍使用 SocketCAN 实现通信时使用的应用程序开发接口。

(1). 初始化

SocketCAN 中大部分的数据结构和函数在头文件 linux/can.h 中进行了定义。 CAN 总线套接字的创建采用标准的网络套接字操作来完成。网络套接字在头文件 sys/socket.h 中定义。 套接字的初始化方法如下：

int s;

struct sockaddr\_can addr;

struct ifreq ifr;

s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);//创建 SocketCAN 套接字

strcpy(ifr.ifr\_name, "can0" );

ioctl(s, SIOCGIFINDEX, &ifr);//指定 can0 设备

addr.can\_family = AF\_CAN;

addr.can\_ifindex = ifr.ifr\_ifindex;

bind(s, (struct sockaddr \*)&addr, sizeof(addr)); //将套接字与 can0 绑定

(2). 数据发送

在数据收发的内容方面， CAN 总线与标准套接字通信稍有不同，每一次通信都采用 can\_ frame 结构体将数据封装成帧。 结构体定义如下：

struct can\_frame {

canid\_t can\_id;//CAN 标识符

\_\_u8 can\_dlc;//数据场的长度

\_\_u8 data[8];//数据

};

can\_id 为帧的标识符， 如果发出的是标准帧， 就使用 can\_id 的低 11 位； 如果为扩展帧， 就使用 0～ 28 位。 can\_id 的第 29、 30、 31 位是帧的标志位，用来定义帧的类型，定义如下：

#define CAN\_EFF\_FLAG 0x80000000U //扩展帧的标识

#define CAN\_RTR\_FLAG 0x40000000U //远程帧的标识

#define CAN\_ERR\_FLAG 0x20000000U //错误帧的标识，用于错误检查

数据发送使用 write 函数来实现。 如果发送的数据帧(标识符为 0x123)包含单个字节(0xAB)的数据，可采用如下方法进行发送：

struct can\_frame frame;

frame.can\_id = 0x123;//如果为扩展帧，那么 frame.can\_id = CAN\_EFF\_FLAG | 0x123;

frame.can\_dlc = 1; //数据长度为 1

frame.data[0] = 0xAB; //数据内容为 0xAB

int nbytes = write(s, &frame, sizeof(frame)); //发送数据

if(nbytes != sizeof(frame)) //如果 nbytes 不等于帧长度，就说明发送失败

printf("Error\n!");

如果要发送远程帧(标识符为 0x123)，可采用如下方法进行发送：

struct can\_frame frame;

frame.can\_id = CAN\_RTR\_FLAG | 0x123;

write(s, &frame, sizeof(frame));

(3). 数据接收

数据接收使用 read 函数来完成，实现如下：

struct can\_frame frame;

int nbytes = read(s, &frame, sizeof(frame));

当然， 套接字数据收发时常用的 send、 sendto、 sendmsg 以及对应的 recv 函数也都可以用于 CAN总线数据的收发。

(4). 错误处理

当帧接收后，可以通过判断 can\_id 中的 CAN\_ERR\_FLAG 位来判断接收的帧是否为错误帧。 如果为错误帧，可以通过 can\_id 的其他符号位来判断错误的具体原因。

错误帧的符号位在头文件 linux/can/error.h 中定义。

(5). 过滤规则设置

在数据接收时，系统可以根据预先设置的过滤规则，实现对报文的过滤。过滤规则使用 can\_filter 结构体来实现，定义如下:

struct can\_filter {

canid\_t can\_id;

canid\_t can\_mask;

};

过滤的规则为：

接收到的数据帧的 can\_id  & mask == can\_id & mask

通过这条规则可以在系统中过滤掉所有不符合规则的报文，使得应用程序不需要对无关的报文进行处理。在 can\_filter 结构的 can\_id 中，符号位 CAN\_INV\_FILTER 在置位时可以实现 can\_id 在执行过滤前的位反转。

用户可以为每个打开的套接字设置多条独立的过滤规则，使用方法如下：

struct can\_filter rfilter[2];

rfilter[0].can\_id = 0x123;

rfilter[0].can\_mask = CAN\_SFF\_MASK; //#define CAN\_SFF\_MASK 0x000007FFU

rfilter[1].can\_id = 0x200;

rfilter[1].can\_mask = 0x700;

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, &rfilter, sizeof(rfilter));//设置规则

在极端情况下，如果应用程序不需要接收报文，可以禁用过滤规则。这样的话，原始套接字就会忽略所有接收到的报文。在这种仅仅发送数据的应用中，可以在内核中省略接收队列，以此减少 CPU 资源的消耗。禁用方法如下：

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, NULL, 0); //禁用过滤规则

通过错误掩码可以实现对错误帧的过滤， 例如：

can\_err\_mask\_t err\_mask = ( CAN\_ERR\_TX\_TIMEOUT | CAN\_ERR\_BUSOFF );

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_ERR\_FILTER, err\_mask, sizeof(err\_mask));

(6). 回环功能设置

在默认情况下， 本地回环功能是开启的，可以使用下面的方法关闭回环/开启功能：

int loopback = 0; // 0 表示关闭, 1 表示开启( 默认)

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_LOOPBACK, &loopback, sizeof(loopback));

在本地回环功能开启的情况下，所有的发送帧都会被回环到与 CAN 总线接口对应的套接字上。 默认情况下，发送 CAN 报文的套接字不想接收自己发送的报文，因此发送套接字上的回环功能是关闭的。可以在需要的时候改变这一默认行为：

int ro = 1; // 0 表示关闭( 默认), 1 表示开启

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_RECV\_OWN\_MSGS, &ro, sizeof(ro));

### 6.6.2 程序发送示例

http://blog.csdn.net/reille/article/details/49949651

/\* 1. 报文发送程序 canappsend.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <net/if.h>

#include <sys/ioctl.h>

#include <sys/socket.h>

#include <linux/can.h>

#include <linux/can/raw.h>

#ifndef PF\_CAN

#define PF\_CAN 29

#endif

#ifndef AF\_CAN

#define AF\_CAN PF\_CAN

#endif

int main()

{

int s, nbytes;

struct sockaddr\_can addr;

struct ifreq ifr;

struct can\_frame frame[2] = {{0}};

s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW);//创建套接字

strcpy(ifr.ifr\_name, "can0" );

ioctl(s, SIOCGIFINDEX, &ifr); //指定 can0 设备

addr.can\_family = AF\_CAN;

addr.can\_ifindex = ifr.ifr\_ifindex;

bind(s, (struct sockaddr \*)&addr, sizeof(addr));//将套接字与 can0 绑定

//禁用过滤规则，本进程不接收报文，只负责发送

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, NULL, 0);

//生成两个报文

frame[0].can\_id = 0x11;

frame[0]. can\_dlc = 1;

frame[0].data[0] = 'Y';

frame[1].can\_id = 0x22;

frame[1]. can\_dlc = 1;

frame[1].data[0] = 'N';

//循环发送两个报文

while(1)

{

nbytes = write(s, &frame[0], sizeof(frame[0])); //发送 frame[0]

if(nbytes != sizeof(frame[0]))

{

printf("Send Error frame[0]\n!");

break; //发送错误，退出

}

sleep(1);

nbytes = write(s, &frame[1], sizeof(frame[1])); //发送 frame[1]

if(nbytes != sizeof(frame[1]))

{

printf("Send Error frame[1]\n!");

break;

}

sleep(1);

}

close(s);

return 0;

}

### 6.6.3 程序接收示例

/\* 2. 报文过滤接收程序 canappreceive.c \*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <net/if.h>

#include <sys/ioctl.h>

#include <sys/socket.h>

#include <linux/can.h>

#include <linux/can/raw.h>

#ifndef PF\_CAN

#define PF\_CAN 29

#endif

#ifndef AF\_CAN

#define AF\_CAN PF\_CAN

#endif

int main()

{

int s, nbytes;

struct sockaddr\_can addr;

struct ifreq ifr;

struct can\_frame frame;

struct can\_filter rfilter[1];

s = socket(PF\_CAN, SOCK\_RAW, CAN\_RAW); //创建套接字

strcpy(ifr.ifr\_name, "can0" );

ioctl(s, SIOCGIFINDEX, &ifr); //指定 can0 设备

addr.can\_family = AF\_CAN;

addr.can\_ifindex = ifr.ifr\_ifindex;

bind(s, (struct sockaddr \*)&addr, sizeof(addr)); //将套接字与 can0 绑定

//定义接收规则，只接收标识符等于0x11 的报文

rfilter[0].can\_id = 0x11;

rfilter[0].can\_mask = CAN\_SFF\_MASK;

//设置过滤规则

setsockopt(s, SOL\_CAN\_RAW, CAN\_RAW\_FILTER, &rfilter, sizeof(rfilter));

while(1)

{

nbytes = read(s, &frame, sizeof(frame)); //接收报文

//显示报文

if(nbytes > 0)

{

printf(“ID=0x%X DLC=%d data[0]=0x%X\n”, frame.can\_id,

frame.can\_dlc, frame.data[0]);

}

}

close(s);

return 0;

}

### 6.6.4 发送接收测试

1）发送

以下用iproute2在开发板上操作打开CAN设备。

配置并启动设备

[root@Loongson:/can\_tools]#./ip link set can0 up type can bitrate 250000

sja1000\_platform sja1000\_platform.0: setting BTR0=0x23 BTR1=0x04

或者用canutils打开CAN设备。

[root@Loongson:/can\_tools]#./canconfig can0 bitrate 250000

sja1000\_platform sja1000\_platform.0: setting BTR0=0x23 BTR1=0x04

can0 bitrate: 250000, sample-point: 0.857

[root@Loongson:/can\_tools]#./canconfig can0 start

can0 state: ERROR-ACTIVE

将2个开发板的CANH、 CANL联接在一起，分别下载canappsend、canappreceive并运行：

[root@Loongson:/app]#chmod u+x canappsend

[root@Loongson:/app]#./canappsend

接收程序过滤接收ID=0x11的报文：

[root@Loongson:/app]#chmod u+x canappreceive

[root@Loongson:/app]#./canappreceive

ID=0x11 DLC=1 data[0]=0x59

所以只收到以下报文：’Y’=0x59

frame[0].can\_id = 0x11;

frame[0]. can\_dlc = 1;

frame[0].data[0] = 'Y';

以下报文收不到：

frame[1].can\_id = 0x22;

frame[1]. can\_dlc = 1;

frame[1].data[0] = 'N';