目录

[1 libusb 移植 1](#_Toc535140394)

[1.1 在ubuntu上面安装libusb并编译程序： 1](#_Toc535140395)

[1.2 移植libusb到arm： 1](#_Toc535140396)

[2 libusb基础 2](#_Toc535140397)

[3 libusb 实践 11](#_Toc535140398)

[3.1 hotplug 测试并打印描述符 11](#_Toc535140399)

[3.2中断传输 16](#_Toc535140400)

[3.2.1中断传输代码 16](#_Toc535140401)

[3.2.2实验： 20](#_Toc535140402)

[3.3 批量传输读取GPS数据 21](#_Toc535140403)

[3.3.1 GPS数据包解析 21](#_Toc535140404)

[3.3.2 批量传输代码 23](#_Toc535140405)

[3.3.3 实验 28](#_Toc535140406)

[4 问题汇总 30](#_Toc535140407)

[4.1 ARM板热插拔程序无法打开设备： 30](#_Toc535140408)

[4.2 ubuntu 无法打开设备问题 31](#_Toc535140409)

[4.3 ubuntu编译程序找不到数学库问题 31](#_Toc535140410)

# 1 libusb 移植

## 1.1 在ubuntu上面安装libusb并编译程序：

sudo apt-get install libudev-dev

./configure

sudo make

sudo make install

编译：

#include <libusb-1.0/libusb.h>//头文件必须如此包含libusb.h

gcc -o gps\_usb gps\_usb.c -lusb-1.0//编译必须加上-lusb-1.0

chao@chao:~/libusb/libusb-1.0.19/tests$ lsusb

Bus 001 Device 002: ID 0951:1642 Kingston Technology DT101 G2

chao@chao:~/libusb/libusb-1.0.19/tests$ lsusb -d 0951:1642 -v //查看设备的详细信息

注意在ubuntu上面执行程序以root用户身份，否则可能出现打不开设备情况

## 1.2 移植libusb到arm：

新建文件夹libusb，把下载的libusb-1.0.19.tar.bz2和libusb-compat-0.1.5.tar.bz2放在该目录，并解压

1、编译安装libusb

root@chao:/home/chao/libusb/ tar xjf libusb-1.0.19.tar.bz2

root@chao:/home/chao/libusb/ cd libusb-1.0.19/

root@chao:/home/chao/libusb/libusb-1.0.19/

./configure --host=arm-linux --prefix=`pwd`/install CC=arm-linux-gcc --disable-udev

root@chao:/home/chao/libusb/libusb-1.0.19/make

root@chao:/home/chao/libusb/libusb-1.0.19/ make install

2、编译安装libusb-compat(注意安装路径与libusb保持一致，便于不同版本兼容)

root@chao:/home/chao/libusb/libusb-1.0.19/ export PKG\_CONFIG\_PATH=/home/chao/libusb/libusb-1.0.19/install/lib/pkgconfig:$PKG\_CONFIG\_PATH

root@chao:/home/chao/libusb/libusb-1.0.19/ echo $PKG\_CONFIG\_PATH //查看是否配置//成功，为了compat使用

book@book-desktop:/work/nfs\_root/fs\_mini\_mdev\_new/libusb$

tar xjf libusb-compat-0.1.5.tar.bz2

root@chao:/home/chao/libusb/ cd libusb-compat-0.1.5/

root@chao:/home/chao/libusb/libusb-compat-0.1.5/ ./configure --host=arm-linux CC=arm-linux-gcc --prefix=`pwd`/install

root@chao:/home/chao/libusb/libusb-compat-0.1.5/ make

root@chao:/home/chao/libusb/libusb-compat-0.1.5/ make install

3、拷贝库和头文件

root@chao:/home/chao/libusb mkdir libusb

root@chao:/home/chao/libusb# mkdir libusb/include /libusb/lib/

root@chao:/home/chao/libusb/libusb-compat-0.1.5/install# cp include/\* ../../libusb/include/

root@chao:/home/chao/libusb/libusb-compat-0.1.5/install# cp lib/\* ../../libusb/lib/

root@chao:/home/chao/libusb/libusb-1.0.19/install# cp include/libusb-1.0/\* ../../libusb/include

root@chao:/home/chao/libusb/libusb-1.0.19/install# cp lib/\* ../../libusb/lib/

4、拷贝库到板子

先将lib目录拷贝到work

板子上执行

cd /mnt/lib/

cp \* /usr/lib

5、交叉编译程序：

#include <libusb-1.0/libusb.h>改为#include <libusb.h>

arm-linux-gcc test.c -o test -I/home/chao/libusb/libusb/include -L/home/chao/libusb/libusb/lib -lusb-1.0

# 2 libusb基础

**1 libusb\_init**

函数原型：int libusb\_init(libusb\_context \*\*ctx);

功能说明：该函数进行libusb的初始化，必须最先调用。

参数说明：ctx通常设置NULL

返回值：0成功，非0 失败

**2 libusb\_exit**

函数原型：void libusb\_exit(libusb\_context \*ctx);

功能说明：和libusb\_init成对使用，释放相应的资源。

参数说明：ctx通常设置NULL

**3 libusb\_has\_capability**

函数原型：int libusb\_has\_capability(uint32\_t capability);

功能说明：判断当前的库是否支持某项功能

参数说明：capability的取值范围在 enum libusb\_capability中定义。

LIBUSB\_CAP\_HAS\_CAPABILITY libusb库的 API是否有效，该项通常总是返回1

LIBUSB\_CAP\_HAS\_HOTPLUG 是否支持热插拔

LIBUSB\_CAP\_HAS\_HID\_ACCESS 是否支持访问HID设备，而不需要用户干预

LIBUSB\_CAP\_SUPPORTS\_DETACH\_KERNEL\_DRIVER 是否支持在内核中释放默认的驱动也就是可以调用libusb\_detach\_kernel\_driver来释放内核驱动

返回值：非0 支持 ，0 不支持

**4 libusb\_hotplug\_register\_callback**

函数原型：int LIBUSB\_CALL libusb\_hotplug\_register\_callback(libusb\_context \*ctx, libusb\_hotplug\_event events,libusb\_hotplug\_flag flags, int vendor\_id, int product\_id, int dev\_class,libusb\_hotplug\_callback\_fn cb\_fn,void \*user\_data,libusb\_hotplug\_callback\_handle \*handle);

功能说明：注册回调函数，响应热插拔事件。

参数说明：ctx 通常为NULL

events 要响应的事件，参数为LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED设备插入事件 LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_LEFT 设备拔出事件， 也可以LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED|LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_LEFT表示同时响应插拔事件

flags 如果为0，只有发生插拔的时候才会调用注册的回调函数，如果为LIBUSB\_HOTPLUG\_ENUMERATE，则在初始化前设备已经插入，也会调用注册的回掉函数。

vendor\_id 需要监控的VID，只有指定的VID的设备插拔，才会调用回调函数 。设置为LIBUSB\_HOTPLUG\_MATCH\_ANY，则不判断VID

product\_id 需要监控的PID，如果设置为LIBUSB\_HOTPLUG\_MATCH\_ANY，则不判断PID

dev\_class 需要监控的设备class,如果设置为LIBUSB\_HOTPLUG\_MATCH\_ANY，则不判断class。注意这里的class是与libusb\_device\_descriptor的class匹配，而不是 libusb\_interface\_descriptor的class

cb\_fn 回调函数的指针 回调函数的定义为int LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, void \*user\_data)

user\_data 用户数据的指针 对应回调函数中的user\_data

handle 句柄

返回值：0成功，非0 失败

注意：不要在回调函数中调用可能阻塞的操作，否则可能造成libusb的其他函数执行失败，不要在回调函数中调用libusb\_claim\_interface等操作，有可能会失败

**5 libusb\_hotplug\_deregister\_callback**

函数原型：void LIBUSB\_CALL libusb\_hotplug\_deregister\_callback(libusb\_context \*ctx,libusb\_hotplug\_callback\_handle handle);

参数说明：注销libusb\_hotplug\_register\_callback函数注册的回调函数

参数说明：ctx 通常为NULL

handle libusb\_hotplug\_register\_callback返回的句柄

返回值：无

**6 libusb\_handle\_events**

函数原型： int LIBUSB\_CALL libusb\_handle\_events(libusb\_context \*ctx);

功能说明：在阻塞模式中处理任何挂起的事件。

参数说明：ctx 通常为NULL

返回值：0成功，非0 失败

注意：1调用libusb\_handle\_events的线程，不要执行阻塞的操作，否则可能造成libusb其他函数执行失败

2 如果注册了热插拔事件，必须要在循环中调用这个函数 例如下面的代码如论怎么插拔USB设备，都不会打印“device insert”

static int LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, void \*user\_data)

{

printf("device insert \n");

}

int main(int argc, char \*\*argv)

{

libusb\_hotplug\_callback\_handle hp;

libusb\_init (NULL);

libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED, LIBUSB\_HOTPLUG\_ENUMERATE, LIBUSB\_HOTPLUG\_MATCH\_ANY,

LIBUSB\_HOTPLUG\_MATCH\_ANY, 0, hotplug\_callback, NULL, &hp);

while(1);

libusb\_hotplug\_deregister\_callback(NULL,hp);

}

必须要改为下面的代码,插拔USB才会有“device insert” 的信息

static int LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, void \*user\_data)

{

printf("device insert \n");

}

int main(int argc, char \*\*argv)

{

libusb\_hotplug\_callback\_handle hp;//句柄

libusb\_init (NULL);

libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED, LIBUSB\_HOTPLUG\_ENUMERATE, LIBUSB\_HOTPLUG\_MATCH\_ANY,

LIBUSB\_HOTPLUG\_MATCH\_ANY, 0, hotplug\_callback, NULL, &hp);

while(1)

{

libusb\_handle\_events(NULL);

}

libusb\_hotplug\_deregister\_callback(NULL,hp);

}

**7 libusb\_open\_device\_with\_vid\_pid**

函数原型：libusb\_device\_handle \* LIBUSB\_CALL libusb\_open\_device\_with\_vid\_pid( libusb\_context \*ctx, uint16\_t vendor\_id, uint16\_t product\_id);

函数功能：通过VID和PID打开一个USB 设备，并返回设备句柄libusb\_device\_handle的指针

参数说明：ctx 通常为NULL

vendor\_id 设备的VID

product\_id 设备的PID

返回值：成功返回libusb\_device\_handle的指针 ，失败返回NULL

**8 libusb\_open**

函数原型：int LIBUSB\_CALL libusb\_open(libusb\_device \*dev, libusb\_device\_handle \*\*handle);

函数功能：通过libusb\_device的指针打开一个USB设备，并返回设备句柄libusb\_device\_handle的指针

参数说明：dev libusb\_device的指针

handle 用来返回设备句柄libusb\_device\_handle的指针

返回值：0成功，非0 失败

**9 libusb\_close**

函数原型：void LIBUSB\_CALL libusb\_close(libusb\_device\_handle \*dev\_handle);

函数功能：关闭 libusb\_open或者libusb\_open\_device\_with\_vid\_pid打开的设备

参数说明：dev\_handle 调用libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的设备句柄libusb\_device\_handle的指针

返回值： 无

**10 libusb\_get\_device\_list**

函数原型：ssize\_t LIBUSB\_CALL libusb\_get\_device\_list(libusb\_context \*ctx, libusb\_device \*\*\*list);

函数功能：获取当前的设备列表

函数功能：ctx 通常为NULL

list USB设备列表

返回值：0成功，非0 失败

**11 libusb\_free\_device\_list**

函数原型：void LIBUSB\_CALL libusb\_free\_device\_list(libusb\_device \*\*list, int unref\_devices);

函数功能：释放以前使用的设备列表

参数说明：list 要释放的设备列表的指针

unref\_devices 如果该参数置1 列表中的每个设备的引用计数减1

返回值：无

注：下面是示例代码：

libusb\_device \*\*devs;

ssize\_t cnt;

int i;

libusb\_init (NULL);

cnt = libusb\_get\_device\_list(NULL, &devs);

for(i=0;i<cnt;i++)

{

PrintUsbDec(devs[i]);

}

libusb\_free\_device\_list(devs, 1);

**12 libusb\_get\_device\_descriptor**

函数原型：int LIBUSB\_CALL libusb\_get\_device\_descriptor(libusb\_device \*dev, struct libusb\_device\_descriptor \*desc);

函数功能：获取USB设备的设备描述符

参数说明：dev libusb\_device的指针，是要读取的设备

desc 设备描述符的指针，用来带回设备描述符的结构

返回值：0成功，非0 失败

注意：这个函数是将设备描述符的结构拷贝到desc指向的地址

所以,下面的用法是错误的

struct libusb\_device\_descriptor \*desc;

libusb\_get\_device\_descriptor(dev,desc);

正确的写法是

struct libusb\_device\_descriptor desc;

libusb\_get\_device\_descriptor(dev,&desc);

**13 libusb\_get\_config\_descriptor**

函数原型：int LIBUSB\_CALL libusb\_get\_config\_descriptor(libusb\_device \*dev, uint8\_t config\_index, struct libusb\_config\_descriptor \*\*config);

函数功能：获取指定设备的配置描述符

参数说明：dev libusb\_device的指针，是要读取的设备

config\_index 配置描述符的索引（一个USB设备可能有多个配置）

config 配置描述符的指针，用来带回设备描述符

返回值：0成功，非0 失败

**14 libusb\_free\_config\_descriptor**

函数原型：void LIBUSB\_CALL libusb\_free\_config\_descriptor( struct libusb\_config\_descriptor \*config);

函数功能：释放配置描述符

参数说明：config 要释放的配置描述符

返回值：无

注意： 用libusb\_get\_config\_descriptor获取配置描述符后必须要调用 libusb\_free\_config\_descriptor释放，否则会造成内存泄漏。

**15 libusb\_control\_transfer**

函数原型：int LIBUSB\_CALL libusb\_control\_transfer(libusb\_device\_handle \*dev\_handle, uint8\_t request\_type, uint8\_t bRequest, uint16\_t wValue, uint16\_t wIndex,

unsigned char \*data, uint16\_t wLength, unsigned int timeout);

参数说明：dev\_handle libusb\_device\_handle的指针

request\_type D7=0主机到设备， =1设备到主机；

D6D5 =00标准请求命令， 01 类请求命令，10用户定义的命令，11保留值

D4D3D2D1D0= 0表示接收者为设备，1表示接收者为接口，2表示接收者为端点，3表示接收者为其他，其他值保留

这个参数由 enum libusb\_request\_recipient 、enum libusb\_request\_type 、enum libusb\_endpoint\_direction 组合

bRequest 命令的序号(其实就是命令)；所有的命令都是以不同编码值的方式传递给设备的，bRequest就表示USB命令的编码值。可以是USB标准命令，也可以用户自定义命令

标准请求命令定义在 enum libusb\_standard\_request

Value 2个字节，用来传送当前请求的参数，随请求不同而变。

Index 索引字段同样是2个字节，描述的是接口号

data 要传输的数据的指针，不需要传输数据，设置为NULL

wLength 数据的长度。当命令不需要传输数据时，此字段设为0

timeout 设置超时的毫秒数，如果设置0，则永不超时

注：USB标准请求命令：

1.获取状态 Get Status (00H)

A:[To Device]获取设备的状态:

\*.位0:自供电(0表示总线供电;1表示自供电).

\*.位1:远程唤醒(0表示不支持远程唤醒;1表示远程唤醒).

\*.位2~15:保留.

\*.一般选择总线供电,不支持远程唤醒,所以返回数据就是0x0000.

B:[To Interface]获取接口的状态:

\*.接口状态的16位字节全部保留,所以返回数据就是0x0000.

C:[To Endpoint]获取端点的状态:

\*.位0:Halt(0表示端点允许;1表示端点禁止).

\*.位1~15:保留(复位为0).

2.清除特性 Clear Feature (01H)

A:[To Device]清除设备的远程唤醒功能,并返回一个空包.

B:[To Endpoint]解禁端点.

3.设置特性 Set Feature (03H)

A:[To Device]设置设备的远程唤醒功能,并返回一个空包.

B:[To Endpoint]禁止端点.

4.设置地址 Set Address (05H)

A:设置设备地址.

5.获取描述符 Get Descriptor (06H)

A:[To Device]获取设备描述符:

\*.描述当前USB协议的版本号.设备端点0的FIFO大小.USB设备的ID号等.

B:[To Configuration]获取配置描述符:

\*.描述USB设备接口个数及是否有自供电能力等.

C:[To Interface]获取接口描述符:

\*.描述端点0以外的物理端点个数等信息.

D:[To Endpoint]获取端点描述符:

\*.描述端点0各端点的传输类型和最大信息包大小和端点的传输方向(IN/OUT).

6.设置描述符(可选,无法更新) Set Descriptor (07H)

7.获取配置信息 Get Configuration (08H)

8.设置配置 Set Configuration (09H)

A:[To Configuration]设置配置描述符.

B:[To Interface]设置接口描述符.

C:[To Endpoint]设置端点描述符.

9.获取接口信息 Get Interface (0AH)

10.设置接口 Set Interface (0BH)

11.SYNCH\_FRAME(0CH)

用于设备设置和报告一个端点的同步帧.

**16 libusb\_kernel\_driver\_active**

函数原型：int LIBUSB\_CALL libusb\_kernel\_driver\_active(libusb\_device\_handle \*dev,int interface\_number);

函数功能：确定指定接口的内核驱动程序是否已经激活。如果一个内核驱动程序是激活的，libusb\_claim\_interface

调用的会失败

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

interface\_number 接口号，这个对应接口描述符的 bInterfaceNumber

返回值：1 已经激活，非1 没有激活

**17 libusb\_detach\_kernel\_driver**

函数原型：int LIBUSB\_CALL libusb\_detach\_kernel\_driver(libusb\_device\_handle \*dev,int interface\_number);

函数功能：卸载指定接口的内核驱动程序。如果一个内核驱动程序是激活的，必须先调用这个函数，再调用

libusb\_claim\_interface

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

interface\_number 接口号，这个对应接口描述符的 bInterfaceNumber

返回值：0 成功，非0失败

**18 libusb\_claim\_interface**

函数原型：int LIBUSB\_CALL libusb\_claim\_interface(libusb\_device\_handle \*dev, int interface\_number);

函数功能：为指定的设备申请接口

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

interface\_number 接口号，这个对应接口描述符的 bInterfaceNumber

返回值：0 成功，非0失败

**19 libusb\_release\_interface**

函数原型：int LIBUSB\_CALL libusb\_release\_interface(libusb\_device\_handle \*dev, int interface\_number);

函数功能：释放之前为指定的设备申请接口，注意这个函数只是释放接口，不会重新加载内核驱动

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

interface\_number 接口号，这个对应接口描述符的 bInterfaceNumber

返回值：0 成功，非0失败

**20 libusb\_attach\_kernel\_driver**

函数原型：int LIBUSB\_CALL libusb\_attach\_kernel\_driver(libusb\_device\_handle \*dev, int interface\_number);

函数功能：加载指定接口的内核驱动

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

interface\_number 接口号，这个对应接口描述符的 bInterfaceNumber

返回值：0 成功，非0失败

下面是示例代码：

if(libusb\_kernel\_driver\_active(handle, bInterfaceNumber) == 1) //判断内核驱动时候加载

{

if(libusb\_detach\_kernel\_driver(handle, bInterfaceNumber) == 0) //卸载驱动，例如我们操作的是一个U盘，那么执行到这里设备文件里面的U盘将消失

{

printf("Kernel Driver Detached!");

}

}

libusb\_claim\_interface(handle, bInterfaceNumber);

..........对设备进行读写操作

libusb\_release\_interface(handle, bInterfaceNumber); //释放请求的接口

libusb\_attach\_kernel\_driver(handle, bInterfaceNumber); //加载内核驱动，U盘将重新出现在设备文件里

**21 libusb\_set\_auto\_detach\_kernel\_driver**

函数原型：int LIBUSB\_CALL libusb\_set\_auto\_detach\_kernel\_driver( libusb\_device\_handle \*dev, int enable);

函数功能：设置自动卸载内核驱动，注意这个函数调用时不会卸载内核驱动，只是做标记。在调用libusb\_claim\_interface的时候卸载内核驱动

在调用libusb\_release\_interface的时候自动加载内核驱动

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

enable 1 使能自动卸载的功能，0关闭

返回值：0 成功，非0失败

例如 下面的代码和上面的代码效果等同

libusb\_set\_auto\_detach\_kernel\_driver(handle,1);

libusb\_claim\_interface(handle, bInterfaceNumber); //请求接口之前先卸载内核驱动

..........对设备进行读写操作

libusb\_release\_interface(handle, bInterfaceNumber); //释放请求的接口后自动加载内核驱动

**22 libusb\_bulk\_transfer**

函数原型：int LIBUSB\_CALL libusb\_bulk\_transfer(libusb\_device\_handle \*dev\_handle,unsigned char endpoint, unsigned char \*data, int length,int \*actual\_length, unsigned int timeout);

函数功能：执行USB批量传输。该函数可以处理输入和输出，根据端点地址的方向位推断传输方向，该函数采用同步模式，数据传输完毕才返回

参数说明：dev\_handle 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

endpoint 端点地址 最高位为1表示输入

data 发送或者接收缓冲区指针

length 缓冲区长度

actual\_length 带回实际传输长度

timeout 超时的毫秒数，0 永不超时

返回值：0 成功，非0失败

**23 libusb\_clear\_halt**

函数原型：int LIBUSB\_CALL libusb\_clear\_halt(libusb\_device\_handle \*dev, unsigned char endpoint);

函数功能：清除端点的halt/stall状态，libusb\_bulk\_transfer有可能返回LIBUSB\_ERROR\_PIPE，这是需要调用这个函数

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

endpoint 出错的端点地址

返回值：0 成功，非0失败

**24 libusb\_interrupt\_transfer**

函数原型：int LIBUSB\_CALL libusb\_interrupt\_transfer(libusb\_device\_handle \*dev\_handle,unsigned char endpoint, unsigned char \*data, int length,int \*actual\_length, unsigned int timeout);

函数功能：执行USB中断传输。该函数可以处理输入和输出，根据端点地址的方向位推断传输方向，该函数采用同步模式，数据传输完毕才返回

参数说明：dev\_handle 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

endpoint 端点地址 最高位为1表示输入

data 发送或者接收缓冲区指针

length 缓冲区长度

actual\_length 带回实际传输长度

timeout 超时的毫秒数，0 永不超时

返回值：0 成功，非0失败

**25 libusb\_set\_configuration**

函数原型：int LIBUSB\_CALL libusb\_set\_configuration(libusb\_device\_handle \*dev,int configuration);

函数功能：为设备设置一个配置参数，大部分设备只有一个配置这个函数通常不需要调用。当某个USB设备有多个配置的时候需要设置

参数说明：dev 调用 libusb\_open或者libusb\_open\_device\_with\_vid\_pid返回的libusb\_device\_handle的句柄

configuration 配置参数，这个对应的是配置描述符里面的bConfigurationValue

返回值：0 成功，非0失败

# 3 libusb 实践

## 3.1 hotplug 测试并打印描述符

#include <signal.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <libusb.h>

**static** **struct** libusb\_device\_handle \*devh = NULL;

**struct** libusb\_device \*\*devs;

**static** **void** dump\_endpoint(**const** **struct** libusb\_endpoint\_descriptor \*endpoint)

{

**static** **const** **char** \* **const** typeattr[] = {

        "Control",

        "Isochronous",

        "Bulk",

        "Interrupt"

    };

**static** **const** **char** \* **const** syncattr[] = {

        "None",

        "Asynchronous",

        "Adaptive",

        "Synchronous"

    };

**static** **const** **char** \* **const** usage[] = {

        "Data",

        "Feedback",

        "Implicit feedback Data",

        "(reserved)"

};

**static** **const** **char** \* **const** hb[] = { "1x", "2x", "3x", "(?\?)" };

    unsigned **int** wmax=endpoint->wMaxPacketSize;

**printf**("      Endpoint Descriptor:\n"

           "        bLength             %5u\n"

           "        bDescriptorType     %5u\n"

            "        bEndpointAddress     0x%02x  EP %u %s\n"

           "        bmAttributes        %5u\n"

           "          Transfer Type            %s\n"

           "          Synch Type               %s\n"

           "          Usage Type               %s\n"

           "        wMaxPacketSize     0x%04x  %s %d bytes\n"

           "        bInterval           %5u\n",

           endpoint->bLength,

           endpoint->bDescriptorType,

            endpoint->bEndpointAddress,

           endpoint->bEndpointAddress & 0x0f,

            (endpoint->bEndpointAddress & 0x80) ? "IN" : "OUT",

           endpoint->bmAttributes,

           typeattr[endpoint->bmAttributes & 3],

           syncattr[(endpoint->bmAttributes >> 2) & 3],

           usage[(endpoint->bmAttributes >> 4) & 3],

           wmax, hb[(wmax >> 11) & 3], wmax & 0x7ff,

           endpoint->bInterval);

    /\* only for audio endpoints \*/

**if** (endpoint->bLength == 9)

**printf**("        bRefresh            %5u\n"

               "        bSynchAddress       %5u\n",

               endpoint->bRefresh, endpoint->bSynchAddress);

}

**static** **void** dump\_altsetting(**const** **struct** libusb\_interface\_descriptor \*interface){

**const** **struct** libusb\_endpoint\_descriptor \*end\_desc;

**int** m;

**printf**("Interface Descriptor:\n"

    "bLength             %5u\n"

    "bDescriptorType     %5u\n"

    "bInterfaceNumber    %5u\n"

    "bAlternateSetting   %5u\n"

    "bNumEndpoints       %5u\n"

    "bInterfaceClass     %5u\n"

    "bInterfaceSubClass  %5u\n"

    "bInterfaceProtocol  %5u\n",

    interface->bLength, interface->bDescriptorType,

    interface->bInterfaceNumber,interface->bAlternateSetting,

    interface->bNumEndpoints,interface->bInterfaceClass,

    interface->bInterfaceSubClass,interface->bInterfaceProtocol);

    //每个设置多个端点

**for** (m = 0; m < interface->bNumEndpoints; m++)

      {

        end\_desc = &interface->endpoint[m];

           dump\_endpoint(end\_desc );

        }

}

**static** **void** dump\_interface( **const** **struct** libusb\_interface \*interface)

{

**int** i;

    //每个接口有多个设置,可以设置成不同的接口

**for** (i = 0; i < interface->num\_altsetting; i++)

    dump\_altsetting(&interface->altsetting[i]);

}

**static** **struct** libusb\_device \*libusb\_search\_device\_with\_id(libusb\_context \*ctx, uint16\_t vendor\_id, uint16\_t product\_id)

{

**struct** libusb\_device \*found = NULL;

**struct** libusb\_device \*dev=NULL;

**struct** libusb\_device\_descriptor desc;

**size\_t** i = 0;

**int** r;

**if** (libusb\_get\_device\_list(ctx, &devs) < 0)

**return** NULL;

**while** ((dev = devs[i++]) != NULL) {

        r = libusb\_get\_device\_descriptor(dev, &desc);

**if** (r < 0)

**goto** out;

**if** (desc.idVendor == vendor\_id && desc.idProduct == product\_id) {

            found = dev;

**printf**("VID:PID: %04X:%04X\n", desc.idVendor,desc.idProduct);

**break**;

        }

    }

out:

    libusb\_free\_device\_list(devs, 1);

**return** found;

}

**static** **struct** libusb\_device \*find\_device()

{

**struct** libusb\_device \*dev;

    dev= libusb\_search\_device\_with\_id(NULL, 0x0951, 0x1642);

**return** dev;

}

**static** **int** LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**int** r;

    sleep(1);

    dev= find\_device();

**if** (dev) {

**printf**("find device \n");

        r=libusb\_open(dev,&devh);

**if**(r)

**printf**("open device error\n");

**else** **printf**("open device ok\n");

    }

**printf** ("attch device\n");

**return** 0;

}

**static** **int** LIBUSB\_CALL hotplug\_callback\_detach(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**if** (devh)

    {

        libusb\_close (devh);

        devh=NULL;

        dev=NULL;

**printf** ("Device detached\n");

    }

**return** 0;

}

**int** main(**void**)

{

**int** r,k;

    libusb\_hotplug\_callback\_handle hp[2];

**struct** libusb\_device \*dev = NULL;

**struct** libusb\_device\_descriptor dev\_desc;

**struct** libusb\_config\_descriptor \*conf\_desc;

    r = libusb\_init(NULL);

**if** (r < 0) {

**printf**("failed to initialise libusb\n");

**goto** out;

    }

**if** (!libusb\_has\_capability (LIBUSB\_CAP\_HAS\_HOTPLUG)) {

**printf** ("Hotplug capabilites are not supported on this platform\n");

        libusb\_exit (NULL);

    }

**printf** ("Hotplug capabilites are  supported on this platform\n");

    r = libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED,LIBUSB\_HOTPLUG\_ENUMERATE, 0x0951,

        0x1642, LIBUSB\_HOTPLUG\_MATCH\_ANY, hotplug\_callback, NULL, &hp[0]);

**if** (r < 0) {

**printf** ( "Error registering callback 0\n");

        libusb\_exit (NULL);

    }

    r = libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_LEFT, LIBUSB\_HOTPLUG\_ENUMERATE,  0x0951,

        0x1642,LIBUSB\_HOTPLUG\_MATCH\_ANY, hotplug\_callback\_detach, NULL, &hp[1]);

**if** (r < 0) {

**printf** ( "Error registering callback 1\n");

        libusb\_exit (NULL);

    }

 /\* 注释掉的代码是用来支持打印描述符的

    libusb\_get\_device\_descriptor(dev, &dev\_desc);

    printf("VID:PID: %04X:%04X\n", dev\_desc.idVendor, dev\_desc.idProduct);

    printf("num of confs: %d\n", dev\_desc.bNumConfigurations);

    libusb\_get\_config\_descriptor(dev, 0, &conf\_desc);

    printf("num of interfaces: %d\n",conf\_desc->bNumInterfaces);

    for (k = 0 ; k < conf\_desc->bNumInterfaces; k++)

        dump\_interface(&conf\_desc->interface[k]);

//要操作的接口号

//cur\_intfnum=conf\_desc->interface[0].altsetting[0].bInterfaceNumber;

    \*/

**while** (1) {

        libusb\_handle\_events (NULL);

    }

out:

**if** (devh)

    {

        libusb\_close (devh);

        devh=NULL;

        dev=NULL;

    }

     libusb\_hotplug\_deregister\_callback(NULL,hp[0]);

    libusb\_hotplug\_deregister\_callback(NULL,hp[1]);

    libusb\_exit(NULL);

}

## 3.2中断传输

### 3.2.1中断传输代码

**中断传输步骤**

1、先申请传输结构体irq\_transfer，由libusb分配

2、调用libusb\_fill\_interrupt\_transfer填充irq\_transfer

3、libusb\_submit\_transfer提交传输请求

4、获取到数据调用注册的回调函数

5、回调函数中获取数据并重新提交传输请求

//实例

#include <signal.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <libusb.h>

#include <memory.h>

#include <errno.h>

**static** **struct** libusb\_device\_handle \*devh = NULL;

**struct** libusb\_device \*\*devs;

**static** **struct** libusb\_transfer \*irq\_transfer = NULL;

**static**  unsigned **char** irqbuf[16];

//传输完成中断函数，收到数据后触发

**static** **void** LIBUSB\_CALL cb\_irq(**struct** libusb\_transfer \*transfer)

{

**int** i;

// 正常读取到数据时返回状态LIBUSB\_TRANSFER\_COMPLETED

**if** (transfer->status != LIBUSB\_TRANSFER\_COMPLETED) {

**fprintf**(stderr, "irq transfer status %d?\n", transfer->status);

        irq\_transfer = NULL;

**return**;

}

 //打印收到的鼠标数据

**printf**("datalen:%d\n",transfer->actual\_length);

**for**(i=0;i< transfer->actual\_length;i++)

**printf**("0x%x ",irqbuf[i]);

        //分析鼠标数据

**printf**("\n");

**if**(irqbuf[0]&0x01) **printf**("left key down\n");

**if**(irqbuf[0]&0x02) **printf**("right key down\n");

**if**(irqbuf[1]==0x02) **printf**("cusor move to right\n");

**if**(irqbuf[1]==0xfe) **printf**("cusor move to left\n");

**if**(irqbuf[2]==0xfe) **printf**("cusor move to forward\n");

**if**(irqbuf[2]==0x02) **printf**("cusor move to backward\n");

**if**(irqbuf[3]==0x01) **printf**("whell move to up\n");

**if**(irqbuf[3]==0xff) **printf**("whell move to down\n");

**if** (libusb\_submit\_transfer(irq\_transfer) < 0)

**exit**(1);

}

**static** **struct** libusb\_device \*libusb\_search\_device\_with\_id(libusb\_context \*ctx, uint16\_t vendor\_id, uint16\_t product\_id)

{

**struct** libusb\_device \*found = NULL;

**struct** libusb\_device \*dev=NULL;

**struct** libusb\_device\_descriptor desc;

**size\_t** i = 0;

**int** r;

**if** (libusb\_get\_device\_list(ctx, &devs) < 0)

**return** NULL;

**while** ((dev = devs[i++]) != NULL) {

        r = libusb\_get\_device\_descriptor(dev, &desc);

**if** (r < 0)

**goto** out;

**if** (desc.idVendor == vendor\_id && desc.idProduct == product\_id) {

            found = dev;

            //printf("VID:PID: %04X:%04X\n", desc.idVendor,desc.idProduct);

**break**;

        }

    }

out:

    libusb\_free\_device\_list(devs, 1);

**return** found;

}

**static** **struct** libusb\_device \*find\_device()

{

**struct** libusb\_device \*dev;

    dev= libusb\_search\_device\_with\_id(NULL, 0x04B3, 0x3107);

**return** dev;

}

**static** **int** LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**int** r,i,k,ret;

    unsigned **char** buf[64];

**int** recv\_len;

**struct** libusb\_device \*pdev;

**struct** libusb\_device\_descriptor dev\_desc;

**struct** libusb\_config\_desacriptor \*conf\_desc;

    usleep(100000);

    pdev= find\_device();

**if** (pdev) {

**printf**("find device \n");

        r=libusb\_open(dev,&devh);

**if**(r)

**printf**("open device error\n");

**else**

**printf**("open device ok\n");

**if**(devh!=NULL){

**printf** ("attch device\n");

            libusb\_set\_configuration(devh,0x01);//设定配置，多个配置的设备需要设定

 /\*

      libusb\_get\_config\_descriptor(dev, 0, &conf\_desc);

//当前操作的接口

cur\_intfnum=conf\_desc->interface[0].altsetting[0].bInterfaceNumber;

\*/

//第二个参数是当前接口

            r = libusb\_kernel\_driver\_active(devh,0);

**if** (r == 1)

            {

                libusb\_detach\_kernel\_driver(devh, 0);

**printf**("driver error\n");

            }

            r = libusb\_claim\_interface(devh ,0);

**if** (r < 0) {

**printf**("usb\_claim\_interface error\n");

            }

**printf**("claimed interface\n");

s/\*中断传输步骤

1、先申请传输结构体irq\_transfer，由libusb分配

2、调用libusb\_fill\_interrupt\_transfer填充irq\_transfer

3、libusb\_submit\_transfer提交传输请求

4、获取到数据调用注册的回调函数

5、回调函数中获取数据并重新提交传输请求\*/

            irq\_transfer = libusb\_alloc\_transfer(0);

**if** (!irq\_transfer)

**return** -ENOMEM;

            libusb\_fill\_interrupt\_transfer(irq\_transfer, devh, 0x81, irqbuf,16, cb\_irq, NULL, 0);

            libusb\_submit\_transfer(irq\_transfer);

        }

    }

**return** 0;

}

**static** **int** LIBUSB\_CALL hotplug\_callback\_detach(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**if** (devh)

    {

        libusb\_close (devh);

        devh=NULL;

**printf** ("Device detached\n");

    }

**return** 0;

}

**int** main(**void**)

{

**int** r,i,k,ret;

    unsigned **char** buf[64];

**int** recv\_len;

    libusb\_hotplug\_callback\_handle hp[2];

**int** cur\_intfnum;

    r = libusb\_init(NULL);

**if** (r < 0) {

**printf**("failed to initialise libusb\n");

**goto** out;

    }

**if** (!libusb\_has\_capability (LIBUSB\_CAP\_HAS\_HOTPLUG)) {

**printf** ("Hotplug capabilites are not supported on this platform\n");

        libusb\_exit (NULL);

**exit**(1);

    }

**printf** ("Hotplug capabilites are  supported on this platform\n");

    r = libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED,LIBUSB\_HOTPLUG\_ENUMERATE,0x04B3, 0x3107,

                 LIBUSB\_HOTPLUG\_MATCH\_ANY, hotplug\_callback, NULL, &hp[0]);

**if** (r < 0) {

**printf** ( "Error registering callback 0\n");

        libusb\_exit (NULL);

    }

    r = libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_LEFT, LIBUSB\_HOTPLUG\_ENUMERATE,  0x04B3, 0x3107,LIBUSB\_HOTPLUG\_MATCH\_ANY, hotplug\_callback\_detach, NULL, &hp[1]);

**if** (r < 0) {

**printf** ( "Error registering callback 1\n");

        libusb\_exit (NULL);

    }

**while**(1){

        libusb\_handle\_events (NULL);

    }

out:

**if** (devh)

    {

        libusb\_release\_interface(devh, 0);

        libusb\_attach\_kernel\_driver(devh,0);

        libusb\_close (devh);

    }

     libusb\_hotplug\_deregister\_callback(NULL,hp[0]);

    libusb\_hotplug\_deregister\_callback(NULL,hp[1]);

    libusb\_exit(NULL);

}

### 3.2.2实验：

编译

arm-linux-gcc interrupt\_transfer.c -o interrupt\_transfer -I/home/chao/libusb/libusb/include -L/home/chao/libusb/libusb/lib -lusb-1.0

/mnt/libusb/test # ./interrupt\_transfer

Hotplug capabilites are supported on this platform

find device

open device ok

attch device

claimed interface

0x1 0x0 0x0 0x0

left key down

0x2 0x0 0x0 0x0

right key down

0x0 0xfe 0x0 0x0

cusor move to left

0x0 0x2 0xff 0x0

cusor move to right

0x0 0x0 0xfe 0x0

cusor move to forward

0x0 0x0 0x2 0x0

cusor move to backward

0x0 0x0 0x0 0x1

whell move to up

0x0 0x0 0x0 0xff

whell move to down

不同的鼠标可能不一样，有的是 4 字节。我的thinkpad小鼠标是4字节  
00 00 00 00  
按键 X方向位移 Y方向位移 滚轮

X方向：往左移是负数，往右移是正数。  
Y 方向：往上移是负数，往下移是正数。  
滚 轮：往前滑是正数，往后滑是负数。

之所以坐标值只提供方向是为了软件可以调整灵敏度。每次移动鼠标反应到屏幕上

是多少完全由软件决定

## 3.3 批量传输读取GPS数据

### 3.3.1 GPS数据包解析

GPS 上电后，每隔一定的时间就会返回一定格式的数据，数据格式为：

$信息类型，x，x，x，x，x，x，x，x，x，x，x，x，x每行开头的字符都是‘$’，接着是信息类型，后面是数据，以逗号分隔开。

一行完整的数据如下：

$GPRMC,080655.00,A,4546.40891,N,12639.65641,E,1.045,328.42,170809,,,A\*60

信息类型为：

GPGSV：可见卫星信息

GPGLL：地理定位信息

GPRMC：推荐最小定位信息

GPVTG：地面速度信息

GPGGA：GPS定位信息

GPGSA：当前卫星信息

GPRMC 最小定位信息

数据详解：$GPRMC,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,<10>,<11>,<12>\*hh

<1> UTC 时间，hhmmss(时分秒)格式

<2> 定位状态，A=有效定位，V=无效定位

<3>纬度ddmm.mmmm(度分)格式(前面的0也将被传输)

<4> 纬度半球N(北半球)或S(南半球)

<5>经度dddmm.mmmm(度分)格式(前面的0也将被传输)

<6> 经度半球E(东经)或W(西经)

<7>地面速率(000.0~999.9节，前面的0也将被传输)

<8>地面航向(000.0~359.9度，以真北为参考基准，前面的0也将被传输)

<9> UTC 日期，ddmmyy(日月年)格式

<10>磁偏角(000.0~180.0度，前面的0也将被传输)

<11> 磁偏角方向，E(东)或W(西)

<12>模式指示(仅NMEA01833.00版本输出，A=自主定位，D=差分，E=估算，N=数据无效)

解析内容：

1.时间，这个是格林威治时间，是世界时间（UTC），我们需要把它转换成北京时间（BTC），BTC和UTC差了8个小时，要在这个时间基础上加8个小时。

2. 定位状态，在接收到有效数据前，这个位是‘V’，后面的数据都为空，接到有效数据后，这个位是‘A’，后面才开始有数据。

3. 纬度，我们需要把它转换成度分秒的格式，计算方法：如接收到的纬度是：4546.40891

4546.40891/100=45.4640891可以直接读出45度, 4546.40891–45\*100=46.40891, 可以直接读出46分

46.40891–46 =0.40891\*60=24.5346读出24秒, 所以纬度是：45度46分24秒。

4. 南北纬，这个位有两种值‘N’（北纬）和‘S’（南纬）

5. 经度的计算方法和纬度的计算方法一样

6. 东西经，这个位有两种值‘E’（东经）和‘W’（西经）

7.速率，这个速率值是海里/时，单位是节，要把它转换成千米/时，根据：1海里=1.85公里，把得到的速率乘以1.85。

8. 航向，指的是偏离正北的角度

9. 日期，这个日期是准确的，不需要转换

GPGGA GPS定位数据

数据详解：$GPGGA,<1>,<2>,<3>,<4>,<5>,<6>,<7>,<8>,<9>,M,<10>,M,<11>,<12>\*xx<CR><LF>

$GPGGA：起始引导符及语句格式说明(本句为GPS定位数据)；

<1> UTC 时间，格式为hhmmss.sss；

<2> 纬度，格式为ddmm.mmmm(第一位是零也将传送)；

<3> 纬度半球，N 或S(北纬或南纬)

<4> 经度，格式为dddmm.mmmm(第一位零也将传送)；

<5> 经度半球，E 或W(东经或西经)

<6> 定位质量指示，0=定位无效，1=定位有效；

<7>使用卫星数量，从00到12(第一个零也将传送)

<8>水平精确度，0.5到99.9

<9>天线离海平面的高度，-9999.9到9999.9米M指单位米

<10>大地水准面高度，-9999.9到9999.9米M指单位米

<11>差分GPS数据期限(RTCMSC-104)，最后设立RTCM传送的秒数量

<12>差分参考基站标号，从0000到1023(首位0也将传送)。

解析内容：第9，10 个字段，海平面高度和大地水准面高度，单位是米

GPVTG 地面速度信息

$GPVTG,<1>,T,<2>,M,<3>,N,<4>,K,<5>\*hh

　　<1> 以正北为参考基准的地面航向(000~359度，前面的0也将被传输)

　　<2> 以磁北为参考基准的地面航向(000~359度，前面的0也将被传输)

　　<3> 地面速率(000.0~999.9节，前面的0也将被传输)

　　<4> 地面速率(0000.0~1851.8公里/小时，前面的0也将被传输)

　　<5> 模式指示(仅NMEA0183 3.00版本输出，A=自主定位，D=差分，E=估算，N=数据无效

### 3.3.2 批量传输代码

**批量传输步骤**

1、先申请传输结构体bulk\_transfer，由libusb分配

2、调用libusb\_fill\_bulk\_transfer填充bulk\_transfer

3、libusb\_submit\_transfer提交传输请求

4、获取到数据调用注册的回调函数cb\_bulk

5、回调函数中获取数据并重新提交传输请求

#include <signal.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <libusb-1.0/libusb.h>

#include <memory.h>

#include <errno.h>

**typedef** **struct** \_\_gprmc\_\_

{

    unsigned **int** **time**;           //时间

    unsigned **char** pos\_state;     //定位状态

**float** latitude;             //纬度

**char**   north;

**float** longitude;            //经度

**char** east;

**float** speed;                //移动速度

**float** direction;            //方向

    unsigned **int** date;                  //日期

**float** declination;          //磁偏角

**char** dd;                    //磁偏角方向

**char** mode;

} GPRMC;

**static** **struct** libusb\_device\_handle \*devh = NULL;

**struct** libusb\_device \*\*devs;

**static** **struct** libusb\_transfer \*bulk\_transfer = NULL;

**char** bulkbuf[1024];

**static**  GPRMC gprmc;

 //提取gps数据

**int** gprmc\_analysis(**char** \*buf,GPRMC \*gprmc)

{

**int** i,j;

**char** pbuf[128];

**char** \*ptr=NULL;

**if**(gprmc == NULL)

**return** -1;

//下面代码为了解决连续两个逗号，sscanf无法匹配的问题

**if**(NULL == (ptr = **strstr**(buf,"$GPRMC")))

**return** -1;

**for** (i=0,j=0;i<**strlen**(ptr);i++,j++)

        {

            pbuf[j]=ptr[i];

**if** (ptr[i] == ','&&ptr[i+1] == ',')

             {

               pbuf[++j]='0';

             }

**if** (ptr[i] == '\n')

             {

              pbuf[i] = '\0';

**break**;

             }

        }

**sscanf**(pbuf,"$GPRMC,%d.00,%c,%f,%c,%f,%c,%f,%f,%d,",&(gprmc->**time**),&(gprmc->pos\_state),&(gprmc->latitude),&(gprmc->north),&(gprmc->longitude),&(gprmc->east),&(gprmc->speed),&(gprmc->direction),&(gprmc->date));

**return** 0;

}

**static** **void** LIBUSB\_CALL cb\_bulk(**struct** libusb\_transfer \*transfer)

{

**int** i;

**if** (transfer->status != LIBUSB\_TRANSFER\_COMPLETED) {

**fprintf**(stderr, "bulk transfer status %d?\n", transfer->status);

        libusb\_free\_transfer(transfer);

        bulk\_transfer = NULL;

**return**;

}

//用于测试解析函数

//char \*str="$GPRMC,121437.00,A,3323.09516,N,11359.38493,E,0.098,,110119";

//打印原始数据

    for(i=0;i< transfer->actual\_length;i++)

        printf("%c",bulkbuf[i]);

     gprmc\_analysis(bulkbuf, &gprmc);

**printf**("时间 : %02d:%02d:%02d\n",(gprmc.**time**/10000+8)%24,(gprmc.**time**%10000)/100,gprmc.**time**%100);

**printf**("GPS状态 : %c \n",gprmc.pos\_state);

**printf**("维度:%f\n",(gprmc.latitude/100));

**printf**(" 北半球:%c\n",gprmc.north);

**printf**("经度:%f\n",(gprmc.longitude/100));

**printf**("东半球:%c\n",gprmc.east);

**printf**("速度 : %.3f \n",gprmc.speed);

**printf**("方向 : %.3f \n",gprmc.direction);

**printf**("日期 : 20%02d-%02d-%02d\n", gprmc.date%100, (gprmc.date%10000)/100,gprmc.date/10000);

**if** (libusb\_submit\_transfer(bulk\_transfer) < 0)

**exit**(1);

}

**static** **struct** libusb\_device \*libusb\_search\_device\_with\_id(libusb\_context \*ctx, uint16\_t vendor\_id, uint16\_t product\_id)

{

**struct** libusb\_device \*found = NULL;

**struct** libusb\_device \*dev=NULL;

**struct** libusb\_device\_descriptor desc;

**size\_t** i = 0;

**int** r;

**if** (libusb\_get\_device\_list(ctx, &devs) < 0)

**return** NULL;

**while** ((dev = devs[i++]) != NULL) {

        r = libusb\_get\_device\_descriptor(dev, &desc);

**if** (r < 0)

**goto** out;

**if** (desc.idVendor == vendor\_id && desc.idProduct == product\_id) {

            found = dev;

            //printf("VID:PID: %04X:%04X\n", desc.idVendor,desc.idProduct);

**break**;

        }

    }

out:

    libusb\_free\_device\_list(devs, 1);

**return** found;

}

**static** **struct** libusb\_device \*find\_device()

{

**struct** libusb\_device \*dev;

    dev= libusb\_search\_device\_with\_id(NULL, 0x1546, 0x01A6);

**return** dev;

}

**static** **int** LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**int** r,i,k,ret;

**int** recv\_len;

**struct** libusb\_device \*pdev;

**struct** libusb\_device\_descriptor dev\_desc;

**struct** libusb\_config\_desacriptor \*conf\_desc;

    usleep(100000);

    pdev= find\_device();

**if** (pdev) {

**printf**("find device \n");

        r=libusb\_open(dev,&devh);

**if**(r)

**printf**("open device error\n");

**else**

**printf**("open device ok\n");

**if**(devh!=NULL){

**printf** ("attch device\n");

            //libusb\_set\_configuration(devh,0x01);

            r = libusb\_kernel\_driver\_active(devh,0);

**if** (r == 1)

            {

                libusb\_detach\_kernel\_driver(devh, 0);

**printf**("driver error\n");

            }

            r = libusb\_claim\_interface(devh ,0);

**if** (r < 0) {

**printf**("usb\_claim\_interface error\n");

            }

**printf**("claimed interface\n");

            bulk\_transfer = libusb\_alloc\_transfer(0);

**if** (!bulk\_transfer)

**return** -ENOMEM;

            libusb\_fill\_bulk\_transfer(bulk\_transfer, devh, 0X82, bulkbuf,**sizeof**(bulkbuf), cb\_bulk, NULL, 0);

            libusb\_submit\_transfer(bulk\_transfer);

        }

    }

**return** 0;

}

**static** **int** LIBUSB\_CALL hotplug\_callback\_detach(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**if** (devh)

    {

        libusb\_close (devh);

        devh=NULL;

**printf** ("Device detached\n");

    }

**return** 0;

}

**int** main(**void**)

{

**int** r,i,k,ret;

    libusb\_hotplug\_callback\_handle hp[2];

**int** cur\_intfnum;

    r = libusb\_init(NULL);

**if** (r < 0) {

**printf**("failed to initialise libusb\n");

**goto** out;

    }

**if** (!libusb\_has\_capability (LIBUSB\_CAP\_HAS\_HOTPLUG)) {

**printf** ("Hotplug capabilites are not supported on this platform\n");

        libusb\_exit (NULL);

**exit**(1);

    }

**printf** ("Hotplug capabilites are  supported on this platform\n");

    r = libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_ARRIVED,LIBUSB\_HOTPLUG\_ENUMERATE,0x1546, 0x01A6,

                 LIBUSB\_HOTPLUG\_MATCH\_ANY, hotplug\_callback, NULL, &hp[0]);

**if** (r < 0) {

**printf** ( "Error registering callback 0\n");

        libusb\_exit (NULL);

    }

    r = libusb\_hotplug\_register\_callback (NULL, LIBUSB\_HOTPLUG\_EVENT\_DEVICE\_LEFT, LIBUSB\_HOTPLUG\_ENUMERATE,0x1546, 0x01A6,LIBUSB\_HOTPLUG\_MATCH\_ANY, hotplug\_callback\_detach, NULL, &hp[1]);

**if** (r < 0) {

**printf** ( "Error registering callback 1\n");

        libusb\_exit (NULL);

    }

**while**(1){

        libusb\_handle\_events (NULL);

    }

out:

**if** (devh)

    {

        libusb\_release\_interface(devh, 0);

        libusb\_attach\_kernel\_driver(devh,0);

        libusb\_close (devh);

    }

     libusb\_hotplug\_deregister\_callback(NULL,hp[0]);

    libusb\_hotplug\_deregister\_callback(NULL,hp[1]);

    libusb\_exit(NULL);

}

### 3.3.3 实验

实验条件：在ubuntu虚拟机上测试

root@chao:/home/chao/libusb/test/bulk# gcc bulk\_transfer.c -o bulk\_transfer -lusb-1.0

root@chao:/home/chao/libusb/test/bulk# ./bulk\_transfer

**实验1：获取数据**

Hotplug capabilites are supported on this platform

find device

open device ok

attch device

claimed interface

$GPRMC,093434.00,A,3323.08547,N,11359.37493,E,0.968,,120119,,,A\*7E

$GPVTG,,T,,M,0.968,N,1.793,K,A\*28

$GPGGA,093434.00,3323.08547,N,11359.37493,E,1,10,1.20,57.0,M,-13.2,M,,\*42

$GPGSA,A,3,03,26,23,14,31,27,08,22,16,32,,,2.26,1.20,1.92\*07

$GPGSV,3,1,11,03,19,251,22,08,11,195,23,14,26,151,27,16,71,289,27\*74

$GPGSV,3,2,11,21,06,086,,22,13,231,19,23,38,309,36,26,65,018,33\*72

$GPGSV,3,3,11,27,35,178,28,31,45,077,23,32,07,150,28\*44

$GPGLL,3323.08547,N,11359.37493,E,093434.00,A,A\*6A

**实验2：提取数据**

//改代码如下

char \*str="$GPRMC,121437.00,A,3323.09516,N,11359.38493,E,0.098,,110119";

gprmc\_analysis(str, &gprmc);

Hotplug capabilites are supported on this platform

find device

open device ok

attch device

claimed interface

时间 : 20:14:37

GPS状态位 : A [A:有效状态 V:无效状态]

纬度:33.230952

北半球:N

经度 : 东经:113.593848

东半球:E

速度 : 0.098

方向 : 0.000

日期 : 2019-01-11

时间 : 20:14:37

GPS状态位 : A [A:有效状态 V:无效状态]

纬度:33.230952

北半球:N

经度 : 东经:113.593848

东半球:E

速度 : 0.098

方向 : 0.000

日期 : 2019-01-11

$GPRMC,121437.00,A,3323.09516,N,11359.38493,E,0.098,,110119,,,A\*7A

3323.08547,,11359.37493

3323.09516,,11359.38493

3323.09549,,11359.38105,

0.51444\*0.030=0.015433=1.54cm

根据移动速度判断是否静止，<0.1m/s

GPS坐标转换距离和速计算

# 4 问题汇总

## 4.1 ARM板热插拔程序无法打开设备：

猜测是热插拔函数执行快于设备节点的形成，要加延时，经过实验证明猜测是正确的。

**static** **int** LIBUSB\_CALL hotplug\_callback(libusb\_context \*ctx, libusb\_device \*dev, libusb\_hotplug\_event event, **void** \*user\_data)

{

**int** r;

    sleep(1);

    dev= find\_device();

**if** (dev) {

**printf**("find device \n");

        r=libusb\_open(dev,&devh);

**if**(r)

**printf**("open device error\n");

**else** **printf**("open device ok\n");

    }

**printf** ("attch device\n");

**return** 0;

}

实验结果：

/mnt/libusb/test # ./hotplug

Hotplug capabilites are supported on this platform

VID:PID: 0951:1642

find device

open device ok

attch device

registering callback

usb 1-1: USB disconnect, device number 23

Device detached

## 4.2 ubuntu 无法打开设备问题

需要以root用户权限执行程序，否则设备无法打开

## 4.3 ubuntu编译程序找不到数学库问题

gcc GPS.c -o gps -lusb-1.0 -lm