**嵌入式Linux综合实践项目：**

**汽车电子CAN总线**

**嵌入式Linux驱动及综合应用**

# 实践指导书

目录

[实践指导书 1](#_Toc326603903)

[实践一、CAN总线驱动 3](#_Toc326603904)

[1、实践目的 3](#_Toc326603905)

[2、实践内容 3](#_Toc326603906)

[3、实践原理 3](#_Toc326603907)

[3．1文件与设备的关系 3](#_Toc326603908)

[3．2虚拟文件系统 4](#_Toc326603909)

[3．3设备驱动的加载 5](#_Toc326603910)

[3．4框架驱动开发 5](#_Toc326603911)

[3．5 CAN总线 6](#_Toc326603912)

[3．5 .1 CAN总线特点 6](#_Toc326603913)

[3.5.2 CAN总线的数据格式 7](#_Toc326603914)

[3．6、系统实现架构 8](#_Toc326603915)

[3．7、MCP2510器件功能介绍 9](#_Toc326603916)

[3．8、 CAN总线驱动程序实现架构 10](#_Toc326603917)

[3．9、CAN总线驱动程序模型 11](#_Toc326603918)

[4、实践步骤 12](#_Toc326603919)

[4．1、准备实践平台 12](#_Toc326603920)

[4．2、寄存器地址映射 13](#_Toc326603921)

[4．3、 驱动的入口和出口 14](#_Toc326603922)

[4．4、设置SPI接口 16](#_Toc326603923)

[4．5、 MCP2510的初始化 18](#_Toc326603924)

[4．6、 MCP2510发送数据和接收数据 20](#_Toc326603925)

[4．7、中断函数 21](#_Toc326603926)

[4．8、写函数 21](#_Toc326603927)

[4．9． CAN总线驱动测试程序 22](#_Toc326603928)

[4．10、驱动调试 24](#_Toc326603929)

[4．11、实践报告要求 24](#_Toc326603930)

[4．12、参考资料 24](#_Toc326603931)

[实践二、汽车电子环境下的CAN总线应用 24](#_Toc326603932)

[1、实践目的 24](#_Toc326603933)

[2、实践的内容 24](#_Toc326603934)

[3、实践原理 25](#_Toc326603935)

[3．1 主控单元 26](#_Toc326603936)

[3．2 车门检测单元 26](#_Toc326603937)

[3．3 制动单元 26](#_Toc326603938)

[3．4 发动机故障检测单元 26](#_Toc326603939)

[4、实践步骤 26](#_Toc326603940)

[5，实践报告要求 27](#_Toc326603941)

# 实践一、CAN总线驱动

## 1、实践目的

巩固对嵌入式Linux驱动程序的框架及虚拟文件系统的理解。

## 2、实践内容

1, 为这一程序写一个测试程序，从传感器接收数据。

2，分析例子程序，画出程序的结构，说明程序的活动。

## 3、实践原理

### 3．1文件与设备的关系

在Linux中文件是一个很重要的结构，表1-1就是文件的结构定义。从Linux操作系统的角度，设备是被抽象成为设备文件的，也就是说在程序中对于设备的操作就是对设备文件进行操作。

struct file

{

struct list\_head f\_list; /\*a list of all opened files\*/

struct dentry \*f\_dentry; /\*pointer to dentry\*/

struct file\_operations \*i\_fop;/\*pointer to file operation\*/

unsigned int f\_uid, f\_gid; /\*UID＆GID\*/

unsigned long f\_version; /\*version\*/

struct address\_space i\_mapping; **/**\*buffer queue is in this data structure\*/

……

};

表1-1 struct file的定义

列出/dev/目录下的设备文件如表1-2所示。

|  |
| --- |
| [root@localhost dev]# ls –l  crw------- 1 root root 14, 12 10-28 19:16 adsp  crw------- 1 root root 14, 4 10-28 19:16 audio  crw------- 1 root root 14, 20 10-28 19:16 audio1 |

表1-2 设备文件示例

以C标识属性的为字符型设备，以B为标识属性的为块设备。以Hda1为例，12为主设备号，3为次设备号。所谓主设备号标识设备的应用类型，如硬盘这类设备的主设备号都为12，次设备号表示在这一类设备中该设备的序号。通过主设备与次设备号就可以惟一地标识一个设备。

### 3．2虚拟文件系统

虚拟文件系统（VFS：Virtual File System）是Linux文件系统的接口。Ext2、FAT32等文件系统实现自已的数据组织逻辑，通过这些接口挂载到虚拟文件系统中，虚拟文件系统是其它文件系统加载的框架。因此虚拟文件系统需要有管理这个框架的数据结构，以及支持“上架”“下架”的函数。

VFS的数据结构主要有：超级块（superblock）,Inode,File,dentry,描述每个打开文件或设备的file结构，描述文件系统类型的file\_system\_tyoe结构、记录已安装文件系统信息的vfsmount结构，记录设备驱动程序的chrdevs、blkdevs等。

对VFS的操作主要有：文件系统的注册与安装，在使用文件、目录与设备之前对inode的获取，文件系统缓存的管理。其中文件系统缓存的管理包括目录项缓存、inode缓存与Buffer cache.

在Linux中设备是抽象成设备文件的，因而对于设备的操作实际上是对于文件的操作。嵌入式系统中往往都要写内核模块或设备驱动将设备映射成设备文件，然后在应用程序中按照该驱动程序实现的文件操作函数对该设备进行控制，其中要实现的文件操作函数是虚拟文件系统中struct file\_operation中定义的函数，如表1-3所示。

|  |
| --- |
| struct file\_operations {  struct module \*owner;  loff\_t (\*llseek) (struct file \*, loff\_t, int);  ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);  ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);  unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);  int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);  int (\*mmap) (struct file \*, struct vm\_area\_struct \*);  int (\*open) (struct inode \*, struct file \*);  int (\*flush) (struct file \*);  int (\*release) (struct inode \*, struct file \*);  int (\*fsync) (struct file \*, struct dentry \*, int datasync);  int (\*lock) (struct file \*, int, struct file\_lock \*);  ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long, loff\_t \*);  ssize\_t (\*sendpage) (struct file \*, struct page \*, int, size\_t, loff\_t \*, int);  unsigned long (\*get\_unmapped\_area)(struct file \*, unsigned long, unsigned long, unsigned long, unsigned long);  }; |

表1-3 struct file\_operations的定义

### 3．3设备驱动的加载

设备驱动既可以在系统启动后手动加载，也可以在系统启动后自动加载，还可以写在内核中作为内核的一部分自动启动。

手动加载采用的方法就是常用的insmod,rmmod,lsmod,modprob命令组。

自动加载有两种方式，一种是通过初始启动脚本加载，一种是在其它自启动程序中调用insmod函数自动加载。

## 3．4框架驱动开发

表1-3是一个简单的框架驱动。这个例子说明了Linux驱动程序的一般结构。

其中module\_init与module\_exit这两个函数将创建两个全局的例程供操作系统内核调用，当驱动程序加载时将会调用module\_init，module\_exit将会在驱动卸载时自动调用。

MODULE\_LICENSE用来声明license，如果没有声明这个驱动程序加载到某些Linux内核时就会报错。

MODULE\_DESCRIPTION与MODULE\_AUTHOR分别是对驱动程序的功能以及作者的描述。

module\_param用来声明一个参数为模块外部接口，即可以在模块加载时指定该参数的值，如：insmod cam.ko test=3.

|  |
| --- |
| #include <linux/kernel.h>  #include <linux/module.h>  #include <linux/init.h>  #include <linux/moduleparam.h>  static int test;  module\_param(test, int, 0644);  void hello\_foo(void)  {  printk("Hello\n");  }  EXPORT\_SYMBOL(hello\_foo);  static int \_\_init hello\_init(void)  {  printk(KERN\_INFO "Hello world\n");  return 0;  }  static void \_\_exit hello\_exit(void)  {  printk(KERN\_INFO "Goodbye world\n");  }  MODULE\_LICENSE("GPL");  MODULE\_DESCRIPTION("Test");  MODULE\_AUTHOR("xxx");  module\_init(hello\_init);  module\_exit(hello\_exit); |

表1-4 框架驱动示例

## 3．5 CAN总线

CAN是Controller Area Network(控制局域网)的缩写，是ISO国际标准化的串行通信协议。

在当前的汽车产业中，由于安全性、舒适性、方便性、低公害、低成本的需求，各种各样的电子控制系统被开发了出来。由于这些系统之间通信所用的数据类型及对可靠性的要求不尽相同，由多条总线构成的情况很多，线束的数量也随之增加。为适应“减少线束的数量” 、“通过多个 LAN”进行大量数据的高速通信”的需要，1986 年德国电气商Bosch公司开发出面向汽车的 CAN 通信协议。此后，CAN 通过 ISO11898 及 ISO11519 进行了标准化。

现在CAN 的高性能和可靠性已被认同，被广泛地应用于工业自动化、船舶、医疗设备、工业设备、汽车电子、自动控制、智能大厦、电力系统、安防监控等方面。

### 3．5 .1 CAN总线特点

(1)多主控制

CAN总线是一种多主总线系统。在总线空闲时，所有的单元都可开始发送消息（多主控制）。最先访问总线的单元可获得发送权。多个单元同时开始发送时，发送高优先级 ID 消息的单元可获得发送权。

一个CAN总线构成的单一网络中，理想情况下可以挂接任意多个节点，实际节点数目受到硬件的电气特性所限制。CAN总线能使用多种介质进行传输，如双绞线、光纤或同轴电缆等。总线信号使用差分电压传输，两条信号线静态时为2.5V左右，此时状态表示为逻辑1，也被称为“隐性”。两条信号线电压出现差值时被称为“显性”，此时电压分别为3.5V和1.5V。当显性位和隐性位同时传输时，总线数值为显性。这种特性是CAN总线仲裁的基础。

CAN总线上所有节点都必须具有相同的比特率。CAN协议采用反向不归零制（NRZ）编码，在数据流中不对时钟信号进行编码。因此，接收时钟信号必须由接收节点恢复并与发送器的时钟同步。然而，并非所有器件都要求具有相同的主振荡器时钟频率。对于采用不同振荡器时钟频率的器件, 应通过适当设置波特率预分频比以及每一时间段中的时间份额的数量来对比特率进行调整。标称比特率是在非再同步情况下，理想振荡器中的理想发送器每秒发送的位数。

(2)发送消息格式固定

在 CAN 协议中，所有的消息都以固定的格式发送。总线空闲时，所有与总线相连的单元都可以开始发送新消息。两个以上的单元同时开始发送消息时，根据标识符（Identifier 以下称为 ID）决定优先级。ID 并不是表示发送的目的地址，而是表示访问总线的消息的优先级。两个以上的单元同时开始发送消息时，对各消息 ID 的每个位进行逐个仲裁比较。仲裁获胜（被判定为优先级最高）的单元可继续发送消息，仲裁失利的单元则立刻停止发送而进行接收工作。

(3)系统的裁剪性

与总线相连的单元没有类似于“地址”的信息。因此在总线上增加单元时，连接在总线上的其它单元的软硬件及应用层都不需要改变。

(4)良好的错误处理机制

所有的单元都可以检测错误（错误检测功能）。检测出错误的单元会立即同时通知其他所有单元（错误通知功能）。正在发送消息的单元一旦检测出错误，会强制结束当前的发送。强制结束发送的单元会不断反复地重新发送此消息直到成功发送为止（错误恢复功能）。

(5) 故障隔离

CAN 可以判断出错误的类型是总线上暂时的数据错误（如外部噪声等）还是持续的数据错误（如单元内部故障、驱动器故障、断线等）。当总线上发生持续数据错误时，可将引起此故障的单元从总线上隔离出去。

### 3.5.2 CAN总线的数据格式

CAN一个位时间可以分为四个部分：同步段、传播段、相位缓冲段1和相位缓冲段2。每段时间份额的数目可以通过CAN总线控制器的相关寄存器来设定。时间份额的大小由系统时钟和波特率预分频值BRP决定：。图10-1 说明了CAN总线的一个位时间的组成部分。



图 1-1 CAN总线的一个位时间

·同步段 —— 用于同步 CAN 总线上的各个节点。输入信号的跳变沿就发生在同步段。 该段持续时间为1TQ。

·传播段 ——用于补偿网络中的物理传输延迟时间。延迟时间包括信号在总线上的传播时间以及节点的内部延迟时间。延迟时间计算为发送器到接收器之间的往返时间，加上输入比较器延迟和输出驱动器延迟时间。通过设置 CNF2 寄存器中的PRSEG2:PRSEG0，传播段的长度可编程设定为 1TQ 至 8TQ。

·相位缓冲段1 ——重同步可以暂时延长。重同步发生在报文位流发送期间，每一个隐性位到显性位跳变沿后。

·相位缓冲段2 ——重同步可以暂时缩短。

上述几个部分的设定与CAN总线的同步、仲裁等信息有关，其主要目的就是要求各节点在一定误差范围内保持同步。

CAN 2.0B 技术规范定义定义了3种格式：标准数据帧(Standard Data Frame)、扩展数据帧(Extended Data Frame)以及远程帧（Remote Frame )。一个标准的嵌入式CAN节点应支持标准数据帧和扩展数据帧。

CAN数据帧如图 10-2所示。数据帧由以下7个不同的位域组成：帧起始、仲裁域、控制域、数据域、CRC域、应答域、帧结尾。数据域的长度可以为0 。

CAN_sample

图 1-2 CAN数据帧格式

## 3．6、系统实现架构

如下图1-3所示，主控制器采用MCP2510和S3C2410，其他节点采用89C51和MCP2510，89C51通过P1端口模拟SPI与MCP2510连接通讯。

基于ARM的CAN总线系统结构框图

图1-3 基于ARM的CAN总线系统结构框图

## 3．7、MCP2510器件功能介绍

MCP2510 是一款独立CAN控制器，是为简化连接CAN总线的应用而开发的。其微控制器（MCU：Micro Control Unit）通过SPI接口与器件进行通信。MCU通过使用标准SPI读写命令对MCP2510的所有寄存器进行读写操作。MCP2510提供的SPI指令如下表10-1所示。

|  |  |  |
| --- | --- | --- |
| 指令名称 | 指令格式 | 功能介绍 |
| 复位 | 1100 0000 | 将内部寄存器复位为缺省状态，并将器件设定为配置模式 |
| 读 | 0000 0011 | 从以指定地址起始的寄存器读取数据 |
| 写 | 0000 0010 | 向以指定地址起始的寄存器写入数据 |
| 发送请求 | 1000 0nnn | 设置 TXBnCTRL.TXREQ 位以启动一个或多个发送缓冲器的报文发送 |
| 状态读 | 1010 0000 | 读取 MCP2510的状态（包括发送接收中断标志位和各请求发送位） |
| 位修改 | 0000 0101 | 对指定寄存器进行位修改 |

表 1-5 SPI指令

MCP2510 采用三个发送缓冲器。每个发送缓冲器占据14 字节的 SRAM。标准数据帧有11个字节的SRAM，扩展数据帧有14个字节的SRAM。一般只用到标准数据帧，它的11个寄存器为TXBnSIDH、TXBnSIDL、TXBnDLC和TXBnDm。前面三个字节是要发送目标端的标识符和发送字节的数目，最后八个寄存器是装载要发送的数据。最多可以一次发送8个字节。

MCP2510 具有两个全文接收缓冲器。每个接收缓冲器配备有多个验收滤波器。除上述专用接收缓冲器外，MCP2510 还具有单独的报文集成缓冲器 (MAB) ，可作为第三个接收缓冲器。MAB对接收到的报文进行组合，并将满足验收滤波器条件的报文传送至RXBN 缓冲器。接收缓冲器原理如下图1-4所示。

图10-4接收缓冲区原理

图 1-4 接收缓冲器原理

### 3．8、 CAN总线驱动程序实现架构

CAN总线通讯程序可以分为三个模块来实现：发送任务、接收任务和中断处理任务。

CAN总线驱动程序的结构如图 1-5所示。在驱动程序中首先建立数据发送和接收缓冲区。发送和接收任务只是从发送和接收缓冲区中发送和读取数据。中断处理任务只负责填充或读取缓冲区中的数据。

当有数据到达时，硬件MCP2510将数据填充到自带的接收缓冲区，当接收缓冲区满时，则发起中断，在中断例程中将数据从接收缓冲区转移到内存中。当应用程序接收数据时，如果数据缓冲区中有数据时，则读取接收缓冲区中的数据而返回，如果接收缓冲中没有数据则不读取。

当有数据需要发送时，应用程序将数据写到MCP2510的发送缓冲区中，如果总线处于空闲状态，则将数据由发送缓冲区发送出去。若总线处于不可发送状态，则返回。当MCP2510检查到总线处于空闲状态时，将MCP2510发起中断，由操作系统响应中断，调用发送中断处理函数，从而将数据由发送缓冲区发送出去。

绘图18

图 1-5 CAN总线驱动程序结构图

### 3．9、CAN总线驱动程序模型

为了对CAN总线的系统活动进行良好的描述，需要对CAN在线建模。这里仍采用双层定时Petri Net。

如图1-6，每一个带ta的方框表示一个DTTPN的抽象变迁，细的箭头表示的是控制流，粗线箭头表示数据流，在箭头上带一小圆圈表示这一根条件弧，不消耗其前继令牌也不改变其前继数据。

CAN

图 1-6 CAN总线驱动的DTTPN图

实践平台核心板2410板的处理器是S3C2410。这是ARM920T内核的嵌入式处理器。

## 4、实践步骤

### 4．1、准备实践平台

（1） 打开实践平台，插入核心模块，CAN总线终端传输模块，并将CAN结点按下图进行联接。

（2）按下图将人体感应传感器与CAN总线终端模块联接。

（3） 用万用表测试电源（12V，5V,3.3V）对地的电阻，电阻应当在200欧姆以上。

（4） 用串口线将平台的串口3与PC机相联，打开PC机超级终端或Linux下的minicom，设置波特率为115200，8位数据，1位停止位，无奇偶校验，无流控。

（5） 上电平台。应当可以在PC端看到系统启到，直到出现Linux文件系统。

### 4．2、寄存器地址映射

在Linux驱动程序中对设备寄存器的访问，需要将物理地址映射到内核虚拟地址空间。CAN总线驱动程序用到了SPI控制寄存器SPCON0、SPI引脚控制寄存器SPCON0**、**SPI状态寄存器SPSTA0、SPI波特率预值寄存器SPPRE0、SPI发送寄存器SPTDAT0、SPI接收寄存器SPRDAT0、片选和中断。其地址映射如表1-7所示。

|  |
| --- |
| #define rSPCON0 (\*(volatile unsigned long \*)r\_SPCON0)  #define rSPSTA0 (\*(volatile unsigned long \*)r\_SPSTA0)  #define rSPPIN0 (\*(volatile unsigned long \*)r\_SPPIN0)  #define rSPPRE0 (\*(volatile unsigned long \*)r\_SPPRE0)  #define rSPTDAT0 (\*(volatile unsigned long \*)r\_SPTDAT0)  #define rSPRDAT0 (\*(volatile unsigned long \*)r\_SPRDAT0)  #define rGPECON (\*(volatile unsigned long \*)r\_GPECON)  #define rGPEUP (\*(volatile unsigned long \*)r\_GPEUP)  #define rGPGCON (\*(volatile unsigned long \*)r\_GPGCON)  #define rGPGUP (\*(volatile unsigned long \*)r\_GPGUP)  #define rGPGDAT (\*(volatile unsigned long \*)r\_GPGDAT)  unsigned long r\_SPCON0,r\_SPSTA0,r\_SPPIN0,r\_SPPRE0;  unsigned long r\_GPECON,r\_GPEUP;  unsigned long r\_SPTDAT0,r\_SPRDAT0;  unsigned long r\_GPGCON,r\_GPGUP,r\_GPGDAT;  //地址映射函数  int address\_map(void)  {  //SPI registers  r\_SPCON0 = ioremap(0x59000000,4);  r\_SPSTA0 = ioremap(0x59000004,4);  r\_SPPIN0 = ioremap(0x59000008,4);  r\_SPPRE0 = ioremap(0x5900000C,4);  r\_SPTDAT0 = ioremap(0x59000010,4);  r\_SPRDAT0 = ioremap(0x59000014,4);  //I/O register  r\_GPECON = ioremap(0x56000040,4);  r\_GPEUP = ioremap(0x56000048,4);   r\_GPGCON = ioremap(0x56000060,4);  r\_GPGUP = ioremap(0x56000068,4);  r\_GPGDAT = ioremap(0x56000064,4);   return 0;  } |

表1-7 S3C2410寄存器地址映射

### 4．3、 驱动的入口和出口

在驱动程序初始化时首先初始化S3C2410处理器的中断模式，设置外部中断1(EINT1)为低电平触发，如表1-7所示。在更改中断模式的过程中，为防止有中断意外地被触发，使用一对宏local\_irq\_save(flags)和local\_irq\_restore(flags)来实现关闭和开启中断。它们在include/asm/system.h中被定义。

在Linux系统中使用set\_external\_irq(IRQ\_EINT1, EXT\_FALLING\_EDGE, GPIO\_PULLUP\_DIS) 来设置外部中断，使用request\_irq(IRQ\_EINT1,&can\_interrupt,SA\_INTERRUPT,"can",NULL)来注册中断服务函数。外部中断是靠外部电路来触发的，set\_external\_irq()中第一个参数是用来指定注册哪一个外部中断，第二个参数EXT\_FALLING\_EDGE是设置下降沿有效，第三个参数GPIO\_PULLUP\_DIS是让外接端口上拉高电平。request\_irq注册了can\_interrupt为CAN总线的中断处理函数。

接下来创建CAN总线的字符设备，candev\_fops被定义为：

|  |
| --- |
| struct file\_operations candev\_fops=  {  read :can\_read,  write :can\_write,  ioctl :can\_ioctl,  open :can\_open,  release :can\_release,  fasync :can\_fasync,  }; |

表1-8 candev\_fops的定义

Fasync用来支持异步处理机制。当驱动程序实现了表1-8中的can\_fasync操作（如1-9所示），在应用程序中， 用户采用异步通信的模式。在应用程序中采用异步通信的步骤如下表所示。

|  |
| --- |
| struct fasync\_struct \*can\_async\_queue;  int can\_fasync(int fd, struct file \*filp, int mode)  {  return fasync\_helper(fd,filp,mode,&can\_async\_queue);  } |

表1-9 实现can\_fasync

|  |  |
| --- | --- |
| signal(SIGIO,detect\_sig\_sigio); | 将SIGIO关联一个信号处理函数 |
| fcntl(sensor\_inputfd, F\_SETOWN, pid); | 文件描述符的SIGIO事件由进程号为pid的进程或进程组(当pid为负数时表示进程组，组号为其绝对值)来处理。 |
| flags = fcntl(sensor\_inputfd, F\_GETFL);  fcntl(sensor\_inputfd, F\_SETFL, flags|O\_ASYNC); | 当O\_ASYNC状态量设置好后，任何时候该文件描述符上发生IO操作都会发送SIGIO。 |

表1-10 应用程序中异步通信的步骤

当在驱动程序中调用kill\_fasync(&can\_async\_queue,SIGIO,POLL\_OUT)发送信号给用户程序时，应用程序中由fcntl(sensor\_inputfd, F\_SETOWN, pid)所设置的接受SIGIO的进程就会接收到信号SIGIO，并执行相应的信号处理函数。

|  |
| --- |
| int init\_module(void)  {  unsigned int result;  int flags;  //开启S3c2410的中断1  local\_irq\_save(flags);  set\_external\_irq(IRQ\_EINT1, EXT\_FALLING\_EDGE, GPIO\_PULLUP\_DIS);  disable\_irq(IRQ\_EINT1);     enable\_irq(IRQ\_EINT1);  result = request\_irq(IRQ\_EINT1,&can\_interrupt,SA\_INTERRUPT,"can",NULL);  if (result)  {  printk("Can't get assigned irq %d,result=%d\n",IRQ\_EINT7,result);  return result;  }  local\_irq\_restore(flags);  devfs\_register(NULL,INT\_DEV\_NAME,DEVFS\_FL\_AUTO\_DEVNUM,INT\_DEV\_MAJOR,1,  S\_IFCHR|S\_IRUGO|S\_IWUGO,&candev\_fops,NULL);  //映射物理地址到虚拟地址  address\_map();  //初始化SPI和MCP2510芯片  local\_irq\_save(flags);  Init\_SPI();  Init\_MCP2510();  local\_irq\_restore(flags);  for(i=0;i<10;i++) spi\_tx\_data(0xff);//初始化设备  Write\_2510(CANINTF,0x00);//清除中断标志位  return 0;  }  void cleanup\_module(void)  {  //释放已申请的资源  disable\_irq(IRQ\_EINT1);  free\_irq(IRQ\_EINT1, NULL);  unregister\_chrdev(INT\_DEV\_MAJOR,INT\_DEV\_NAME);  printk("rmmod successfully\n");  } |

表1-10 CAN总线驱动程序的入口和出口

### 4．4、设置SPI接口

S3C2410包含2个SPI，每个都分别有两个8位转移寄存器用于数据的传输和接收。采用SPI接口, S3C2410可发送、接收与外部设备同步的8位数据。在主机端，SPI的发送频率可以通过为SPPREn寄存器设置合适的值来控制。在从机端，SPI采用主机提供的时钟，不需要设置波特率。当编程者向SPTDATn寄存器写数据时，将会检查ENSCK和SPCONn寄存器中的MSTR位是否被置位，如果置位则启动数据发送。

SPI有3种模式：轮询、中断、DMA传输模式。可以通过设置rSPCON0(rSPCON1)来改变发送、接收方式。本章介绍的驱动程序采用轮询方式。

SPI初始化和发送接收数据如表1-11所示，根据S3C2410的芯片手册，其中SPI编程遵循以下基本步骤:

1. 设置波特率寄存器（SPPREn）。

2. 设置SPCONn以选择合适的SPI模式。

3. 写数据0xFF到SPTDATn(1、0)次以初始化从设备。

4. 发送数据：检查发送准备位（REDY）是否为1，如果为1 的话就开始向SPTDATn写数据。

5. 接收数据：SPCONn的 TAGD 位禁用即处于正常模式；向SPTDATn 写0xFF，然后测试接收准备位是否为1，再从读缓存中读取数据。

|  |
| --- |
| void Init\_SPI(void)  {  int i;  rSPPRE0 = 0xff; //2410 波特率;  rSPCON0 = 0x18; //设置轮询模式  for(i = 0 ; i < 10 ; i++)  {  rSPTDAT0 = 0xff;  }  rGPECON |= 0x0a800000;  rGPECON &= (~0x05400000);  rGPEUP |= 0x3800;  rGPGCON |= 0x10000000;  rGPGCON &= (~0x20000000);  rGPGUP &= (~0x4000);  rGPGDAT |=0x4000; //不选中mcp2510  } |

表1-11 SPI初始化

|  |
| --- |
| //通过SPI发送数据给mcp2510  void spi\_tx\_data(unsigned char data)  {  spi\_poll\_done();  rSPTDAT0 = data; //发送数据  spi\_poll\_done();  } |

表1-12 通过SPI向mcp2510 发送数据

由S3C2410向MCP2510写数据的函数及注释如表1-12所示。

|  |
| --- |
| void Write\_2510(unsigned char W\_ADD, unsigned char W\_DATA)  {  //使能片选  enable2510();    udelay(10000);  //发送写命令  spi\_tx\_data(CMD\_WRITE);  //发送寄存器地址  spi\_tx\_data(W\_ADD);  //发送数据；  spi\_tx\_data(W\_DATA);  //禁止片选  disable2510();  } |

表1-12 向MCP2510写数据

S3C2410从MCP2510读数据的函数及其注释如表1-13所示。

|  |
| --- |
| unsigned char Read\_2510(unsigned char R\_ADD)  {  unsigned char buffer;  //使能片选  enable2510();  udelay(10000);  //发送读命令  spi\_tx\_data(CMD\_READ);  //发送寄存器地址  spi\_tx\_data(R\_ADD);  //发送数据0xff  spi\_tx\_data(0xff);  buffer = rSPRDAT0;  //禁止片选  disable2510();  return buffer;  } |

表1-13 从MCP2510读数据的函数

### 4．5、 MCP2510的初始化

MCP2510按照以下步骤进行初始化：

1. 软件复位进入配置模式；
2. 设置CAN总线波特率
3. 关闭中断，设置ID过滤器；
4. 切换MCP2510到正常状态；
5. 清空接收和发送缓冲区；
6. 开启接收缓冲区，开启中断。

通过设置MCP2510的CNF1、CNF2、CNF3三个寄存器，可以设置CAN的波特率。在输入时钟为16MHz的MCP2510平台上，可按照表10-12所列的值来设置CAN总线波特率。 MCP2510初始化如表1-15 。

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CAN波特率 | 同步段 | 传输段 | 相位1 | 相位2 | CNF1 | CNF2 | CNF3 |
| 125kb/s | 1 | 7 | 4 | 4 | 0X03 | 0X9E | 0X03 |
| 250 kb/s | 1 | 7 | 4 | 4 | 0X01 | 0X9E | 0X03 |
| 500 kb/s | 1 | 7 | 4 | 4 | 0X00 | 0X9E | 0X03 |
| 1 Mb/s | 1 | 3 | 2 | 2 | 0X00 | 0X91 | 0X01 |

表1-14 MCP2510的波特率设置

|  |
| --- |
| void Init\_MCP2510(void)  {  unsigned char buffer;  //设置配置模式  Modify\_2510(CANCTRL, 0xe0, 0x80);  Modify\_2510(CANCTRL, 0x07, 0x00);  //进入配置模式  while(1)  {  buffer = Read\_2510(CANSTAT);  printk("CANSTAT register is %x\n",buffer&0xe0);  if((buffer&0xe0) == 0x80)  {  printk("In the Configure Mode\n");  break;  }  }  //设置波特率  MCP2510\_SetBandRate(250);  //设置中断  Write\_2510(CANINTE, 0xa2);  Write\_2510(CANINTF,0x00);//clear interrupt  Modify\_2510(CANCTRL, 0x07, 0x04);  //设置发送寄存器  Write\_2510(TXRTSCTRL,0x00);  Write\_2510(TXB0CTRL,0x00);  //设置本地ID  //Receive buffer 0 will not be used  Write\_2510(RXM0SIDH,0Xff);  Write\_2510(RXM0SIDL,0Xff);  Write\_2510(RXF0SIDH,0Xff);  Write\_2510(RXF0SIDL,0Xff);  Write\_2510(RXF1SIDH,0Xff);  Write\_2510(RXF1SIDL,0Xff);  //设置buffer1的滤波器  Write\_2510(RXM1SIDH,0xff);  Write\_2510(RXM1SIDL,0xe0);  Write\_2510(RXF2SIDH,0x03);  Write\_2510(RXF2SIDL,0x00);  Write\_2510(RXF3SIDH,0xa1);  Write\_2510(RXF3SIDL,0x00);  Write\_2510(RXF4SIDH,0xa2);  Write\_2510(RXF4SIDL,0x00);  Write\_2510(RXF5SIDH,0xa3);  Write\_2510(RXF5SIDL,0x00);  //设置接收控制寄存器  Write\_2510(RXB1CTRL, 0x62); //receive every frame  Write\_2510(RXB0CTRL, 0x00);  //设置进入正常模式  Write\_2510(CANCTRL, 0x00);  //等待进入正常模式  while(1)  {  buffer = Read\_2510(CANSTAT);  printk("CANSTAT register is %x\n",buffer&0xe0);  if((buffer&0xe0) == 0x00)  {  printk("In the Normal Mode\n");  break;  }  }  printk("Init MCP2510 is over!\n");  flag == 0x88;  } |

表1-15 MCP2510初始化

### 4．6、 MCP2510发送数据和接收数据

CAN驱动接收数据是通过中断来实现的，当控制器产生接收中断时，在中断函数中读取各个缓冲区。

如表1-16,MCP2510发送数据的流程为：

1.写数据到MCP2510发送缓冲区

2.向MCP2510发送传输命令

|  |
| --- |
| void can\_data\_send(void)  {  unsigned char length,i;  length = TXdata[2];  //写数据到发送缓冲区  Write\_2510(TXB0SIDH,TXdata[0]);  Write\_2510(TXB0SIDL,TXdata[1]);  Write\_2510(TXB0DLC, TXdata[2]);  rGPGDAT &=(~0x4000); //Select the chip  udelay(100000);//设置上面寄存器需要一些时间  //发送数据  spi\_tx\_data(CMD\_WRITE);  spi\_tx\_data(TXB0D0);//首先发送第一个发送缓冲区地址  for(i = 0; i<length; i++)  {  spi\_tx\_data(TXdata[i+3]);//依次将数据写入发送缓冲区  }  rGPGDAT |=0x4000; //Unselect the chip  SEND\_TXB0();//发送传输指令  } |

如表1-16 MCP2510发送数据

如表1-17，MCP2510接收数据的流程为：

1.接收到接收信号；

2.读取个接收缓冲区。

|  |
| --- |
| void can\_data\_receive(void)  {  unsigned int length,i;  length = Read\_2510(RXB1DLC);  //Select the chip  rGPGDAT &=(~0x4000);  udelay(100000);  spi\_tx\_data(CMD\_READ);  spi\_tx\_data(RXB1D0);// 首先发送第一个接收缓冲区地址  for(i = 0; i<length; i++)  {  spi\_tx\_data(0xff);  RXdata[i] = rSPRDAT0;  }  rGPGDAT |=0x4000; //Unselect the chip  } |

如表1-16 MCP2510接收数据

### 4．7、中断函数

CAN总线驱动程序的中断子函数主要任务：当产生接收中断时，接收缓冲区中的数据，并且发送消息给用户程序，如表1-17 。

|  |
| --- |
| void can\_interrupt(int irq,void \*d,struct pt\_regs \*regs)  {  unsigned char buffer;  disable\_irq(IRQ\_EINT1); //禁止ARM中断  Write\_2510(CANINTE, 0x00); //禁止MCP2510中断  buffer = Read\_2510(CANINTF);  if((buffer&0x02) == 0x02) //判断是不是接收中断  {  can\_data\_receive();  flag = 0xff;  //Modify\_2510(CANINTF,0x02,0x00);  kill\_fasync(&can\_async\_queue,SIGIO,POLL\_OUT);//异步通知用户已接收到数据  }  //恢复中断  SRCPND &= (~0x00000002);  INTPND = INTPND;  Write\_2510(CANINTE, 0xa2);  enable\_irq(IRQ\_EINT1);  Write\_2510(CANINTF, 0x00);//清除中断标志位  } |

表1-17 MCP2510中断函数

### 4．8、写函数

CAN总线驱动写函数主要是先写入发送缓冲区然后再发送，如表1-18 。

|  |
| --- |
| //写函数  ssize\_t can\_write(struct file \*filp,const char \*buf,size\_t count,loff\_t \*f\_pos)  {  int ret = 0;  int i = 0;  dbuf = kmalloc(count\*sizeof(unsigned char) , GFP\_KERNEL); //分配接收空间  copy\_from\_user(dbuf,buf,count);//将用户空间数据拷进内核  for(i = 0; i < count; i++)  {  TXdata[i] = dbuf[i];  }  //Transmit data to Bus  can\_data\_send();  kfree(dbuf);  return ret;  } |

表1-18 总线写函数

### 4．9． CAN总线驱动测试程序

驱动写好后，需要测试它能不能正常工作。需要写一个专用程序测试它所有功能，如表1-19.

|  |
| --- |
| #include<stdio.h>  #include<stdlib.h>  #include<sys/types.h>  #include<sys/stat.h>  #include<sys/ioctl.h>  #include<fcntl.h>  #include<unistd.h>  #include<signal.h>  #include "mcp2510\_ioctl.h"  #define DEVICE\_NAME "/dev/candev"  char \*buf;  int dev;  void sig\_usr()//接收到信号后执行的函数  {  int count,i;  signal(SIGIO,sig\_usr); //继续接收信号  printf("----receive signal1------\n");  count=read(dev,buf,8); //读取接收到的数据  if(count==8)  {  printf("receive 8 Bytes\n");  for(i = 0; i < 8; i++)  printf("buf[%d] is %x\n",i,buf[i]);  }  else printf("read failed! \n");  }  int main()  {  int j,count,i;  int oflag;  char charexit;  struct sigaction sigact,oldact;  char \*TXdata=(char \*)malloc(11\*sizeof(char));  buf=(char \*)malloc(111\*sizeof(char));  //输入发送数据  TXdata[0] = 0x03;  TXdata[1] = 0x00;  TXdata[2] = 0x08;  TXdata[3] = 0x08 ;  TXdata[4] = 0x07 ;  TXdata[5] = 0x06 ;  TXdata[6] = 0x05 ;  TXdata[7] = 0x04 ;  TXdata[8] = 0x03;  TXdata[9] = 0x02 ;  TXdata[10]= 0x01 ;  signal(SIGIO,sig\_usr);//等待信号  dev=open(DEVICE\_NAME,O\_RDWR);  if(dev>=0)  {  fcntl(dev,F\_SETOWN,getpid());//将用户进程号写到设备文件中，让驱动发送信号到  //用户进程  oflag=fcntl(dev,F\_GETFL);  fcntl(dev,F\_SETFL,oflag|FASYNC);  write(dev,TXdata,11);//发送数据  scanf("%c",&charexit);    while(charexit != 'q')scanf("%c",&charexit);//接收’q’退出  }  else printf("Open falsed !\n");  close(dev);  free(buf);  free(TXdata);  return 0;  } |

表1-19 CAN总线测试程序

这是一个CAN总线发送和接收的示例程序。在这里CAN总线工作在正常模式下，首先发送一断数据，然后等待接收信号。同时主程序从控制台接收用户输入的字符串，当用户输入’q’时，程序退出。

### 4．10、驱动调试

将生成的驱动程序模块insmod到系统中，对系统进行调试，并回答以下三个问题。

* + - 问题1： 驱动程序是否可以加载成功？
    - 问题2：在应用程序中是否可以打开设备？
    - 问题3：在应用程序中是否可以读出结点的状态？

## 4．11、实践报告要求

1，记录实践过程，并将改完的源程序及运行结果写入实践报告。

2，画出上述驱动程序各个函数之间的结构，说明其调用关系；并给出Linux驱动程序的基本组成部分。

## 4．12、参考资料

[1] 嵌入式Linux实践教程,罗怡桂，清华大学出版社，2011.6。

[2] CAN 2.0标准，课程目录下。

[3] SJA1000datasheet.pdf，课程目录下。

[4] MCP2510datasheet.pdf，课程目录下。

# 实践二、汽车电子环境下的CAN总线应用

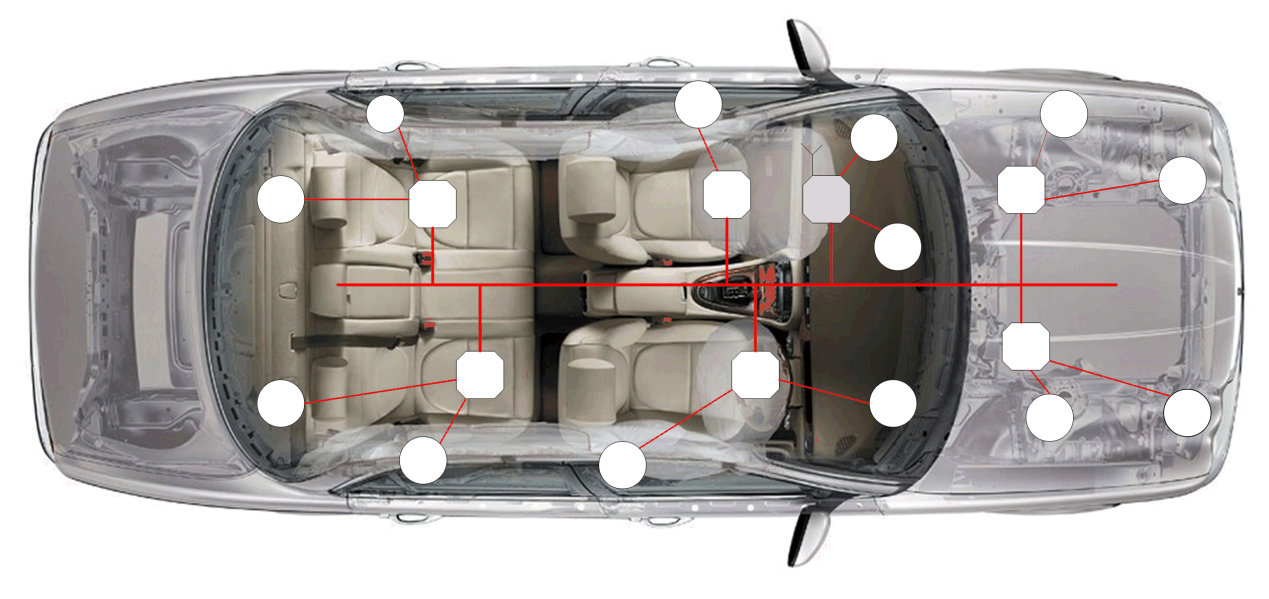
## 1、实践目的

巩固嵌入式驱动与其应用程序之间的关系、掌握嵌入式Linux应用程序的编写

## 2、实践的内容

以汽车电子为应用背景，以实践平台为基础，架构汽车电子CAN总线的模拟应用环境，编写应用程序。

## 3、实践原理





### 3．1 主控单元

接收来自各传感器的数据，判断汽车的工作状况，响应驾驶员的操作指令。满足以下几条规则：

1. 如果车门没有关紧，则发出警报（报警灯闪），不能进行启动操作。如果车子已启动，警报（报警灯闪，3号灯）。
2. 如果制动故障，警报（报警灯闪，2号灯）。
3. 如果发动机故障，警报（报警灯闪，1号灯）。

### 3．2 车门检测单元

实时检测车门的状态，如果车子开始打火启动或车子在运行过程中，如果车门没关则发送给主控单元。

### 3．3 制动单元

如果车子启动或运行，检测到制动故障，则发送警报给主控单元。接收主控单元的制动指令，进行制动（制动灯亮）。

### 3．4 发动机故障检测单元

如果发动机故障，则将故障数据发送给主控单元。

## 4、实践步骤

组织架构如下图：



* 重新分组，每组10个同学，各组选一名组长，一名副组长。
* 选择其中的一个模块，完成该模块的设计与实现。各模块之间的通信协议由组长与副组长在组内讨论完以后，带纸质设计方案，参加中心回桌会议进行讨论定稿。
* 各组完成开发，由两名组长进行协调，将四个实践平台集成起来，进行联调。
* 其中罗老师会要求大家进行交流活动。

## 5，实践报告要求

提交分组讨论纪要

提交各组设计方案、实现代码、单元测试及系统测试记录。