低成本嵌入式 Linux CAN 应用方案

CAN（Controller Area Network）即控制器局域网，由于具有高性能、高可靠性以及简单的网络结构，在工业系统中越来越受到人们的重视，并迅速成为了目前国际上应用最广泛的现场总线之一。

英利嵌入式 Linux 工控主板 EM9260 是一款面向工业自动化领域的高性价比工控板，板上带有标准 CAN 通讯接口。与板上其他标准通讯接口一 样，EM9260 的 CAN 接口实现了相应的嵌入式 Linux 驱动程序，应用程序可以通过打开文件的进行读写的标准方式实现对 CAN 总线接口的数据通讯。本 文侧重于介绍 CAN 通讯方案。

硬件组成

EM9260 嵌入式 Linux 工控板的 CAN 均采用了 PHILIPS 半导体公司的 SJA1000T CAN 总线控制器，SJA1000 是一款独立的控制器，主要用于汽车和一般工业环境中的控制器局域网络（CAN）芯片，它是 PHILIPS 半导体 PCA82C200 CAN 控制器（BasicCAN）的替代产品，而且它增加了一种新的工作模式（PeliCAN），这种模式支持具有很多新特性的 CAN 2.0B 协议。

EM9260 的 CAN 通讯接口可提供高达 1Mbps 的数据传输速率，当采用 5Kbps 的的数据传输速率时其通讯距离最高可达到 10KM。硬件的错误检定特性也增强了 CAN 的抗电磁干扰能力，这给数据的远程可靠传输提供了有利保证。

EM9260 的 CAN 通讯接口根据用户的需要分为两种：一种带光电隔离，一种不带光电隔离。带光电隔离 CAN 总线通讯模块的 CAN 收发器端的所有信号和电 源与其它部分完全隔离，可承受至少 1Kv（有效值）的电压冲击。光电隔离的功能

可在 EM9260 的应用底板上来实现，英利公司在 EM9260 评估底板上提 供了相应的参考电路。

CAN 驱动接口函数

1、CAN 报文的帧格式简介

在 CAN2.0B 中存在两种不同的帧格式，其主要的区别在于标识符的长度，具 11 位标识符的帧称为标准帧，而包括有 29 位标识符的帧称为扩展帧。下面分别介绍数据帧的格式。

1、CAN2.0B 标准帧

CAN 标准帧信息为 11 个字节，包括两部分：信息和数据部分。前 3 个字节为信息部分，如图所示：

	D7	D6	D5	D4	D3	D2	D1	D0
字节 1	FF	RTR	X	X	DLC（数据长度）			
字节 2	（报文识别码）ID.10~ID.3							
字节 3	ID.2~ID.0			RTR				
字节 4	数据 1							
字节 5	数据 2							
字节 6	数据 3							
字节 7	数据 4							
字节 8	数据 5							
字节 9	数据 6							
字节 10	数据 7							
字节 11	数据 8							

注：1、字节 1 为帧信息。D7 位表示帧格式，在标准帧中，FF=0；D6 位表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧，在一般的数据通讯中，只使用数据帧；DLC 表示数据帧实际的数据长度

2、字节 2、字节 3 为报文识别码，11 位有效

3、字节 4～字节 11 为数据帧的实际数据，远程帧时无效

2、CAN2.0B 扩展帧

CAN 标准帧信息为 13 个字节，包括两部分：信息和数据部分。前 5 个字节为信息部分，如图所示：CAN 标准帧信息为 13 个字节，包括两部分：信息和数据部分。前 5 个字节为信息部分，如图所示：

	D7	D6	D5	D4	D3	D2	D1	D0
字节 1	FF	RTR	X	X	DLC（数据长度）			
字节 2	（报文识别码）ID.28~ID.21							
字节 3	ID.20~ID.13							
字节 4	ID.12~ID.5							
字节 5	ID.4~ID.0							
字节 6	数据 1							
字节 7	数据 2							
字节 8	数据 3							
字节 9	数据 4							
字节 10	数据 5							
字节 11	数据 6							
字节 12	数据 7							
字节 13	数据 8							

注：1、字节 1 为帧信息。D7 位表示帧格式，在扩展帧中，FF=1；D6 位表示帧的类型，RTR=0 表示为数据帧，RTR=1 表示为远程帧；DLC 表示数据帧实际的数据长度

2、字节 2～字节 5 为报文识别码，29 位有效

3、字节 6～字节 13 为数据帧的实际数据，远程帧时无效

2、CAN 应用数据结构

英利公司提供的基于嵌入式 Linux 下的 CAN 操作 API 函数，为了方便用户的使用，结合目前常用的一些方法，对于 CAN 接口接收的数据报文采用了以下结构。

```
struct can_frame
    {
        canid_t can_id; /* 用于定义 CAN 报文 ID 以及 EFF/RTR/ERR 等标志 */
        __u8 can_dlc; /* 用于定义 can 报文数据包长度 0-8 */
        __u8 data[8];  /* 用于定义 can 报文数据 */
    };
```

其中的 can 报文 ID 为一个 32 bit 大小的结构，其中各个 bit 位定义如下：

```
typedef __u32 canid_t;
bit 0-28: CAN 报文的 id(标准帧 11bit/扩展帧为 29bit).
bit 29 : CAN 报文错误帧标志(0 = data frame, 1 = error frame)
bit 30 : CAN 报文远程帧标志( 1 = rtr frame )
bit 31 : CAN 报文帧格式标志 (0 = 标准帧, 1 = 扩展帧 )
```

在 进行 CAN 通讯时需要设置相关的参数，包括波特率、选取的数据滤波方式等，其中对于滤波器的设置，在滤波器的作用下，只有当接收报文中的标识位和验收滤波 器预定义的位值相等时，CAN 控制器才允许将收到的报文存入 RXFIFO 中。为了方便使用，在英利公司的 API 函数中采用了一个 struct accept_filter 用来设置相关验收滤波器的相关定义。

```
struct accept_filter
    {
        unsigned int accept_code; /* 用于定义 CAN 报文验收代码位 32bit*/
        unsigned int accept_mask; /* 用于定义 CAN 报文验收屏蔽位 32bit*/
        unsigned char filter_mode; /* 用于定义 CAN 报文滤波模式 */
    };
```

3、CAN 通讯接口 API 函数

EM9260 的系统内核中实现了 CAN 接口的驱动，实现 CAN 接口 open( ) / close() 、read( ) / write( )等函数操作。和在 Linux 下操作设备的方式和操作文件的方式一样，调用 open( )打开设备文件，再调用 read( )、write( )对 CAN 接口进行数据读写操作。另外在此驱动程序的基础上，封装了一套简单实用的API 函数,以满足对于CAN接口一些特殊参数设置的需要。各个函数的 定义在 can_api.h 文件下,在该头文件中对于各个 API 函数均有相应的中文说明。

具体在进行应用程序开发时，首先调用 CAN 接口的 open( )函数打开 CAN 接口：

```
sprintf( portname, '/dev/em9x60_can%d', CanNo );
m_fd = open(portname, O_RDWR |O_NONBLOCK );
```

得到有效的文件描述符 m_fd 后，然后可调用 can_api.h 文件中定义的 API 函数对 CAN 接口进行相应的通讯参数设置：

```
CAN_StartChip( m_fd );
```

```
CAN_SetBaudRate( m_fd, baudrate );
CAN_SetGlobalAcceptanceFilter( m_fd, AcceptanceFilter );
```

再调用 read()/ write()实现 CAN 数据的收发操作。

4、CAN 通讯接口的数据收发应用示例

在英利公司提供的 CAN 方案中，CAN 通讯的数据收发均采用的中断方式，驱动程序中已自动完成了数据的收发，以及内部定义的 CAN 接收缓冲区和发送缓冲区 的管理。对于用户开发应用程序来说，只需要调用英创公司提供的 CAN 通讯 API 函数中的收发函数即可。本小节主要介绍一个 CAN 通讯的综合应用示例程序。

app_cantest 是一个支持 CAN 数据通讯的示例，该例程采用了面向对象的 C++ 编程，把 CAN 数据通讯作为一个对象进行封装，用户调用该对象提供的接口函数即可方便地完成 CAN 数据通讯的操作。

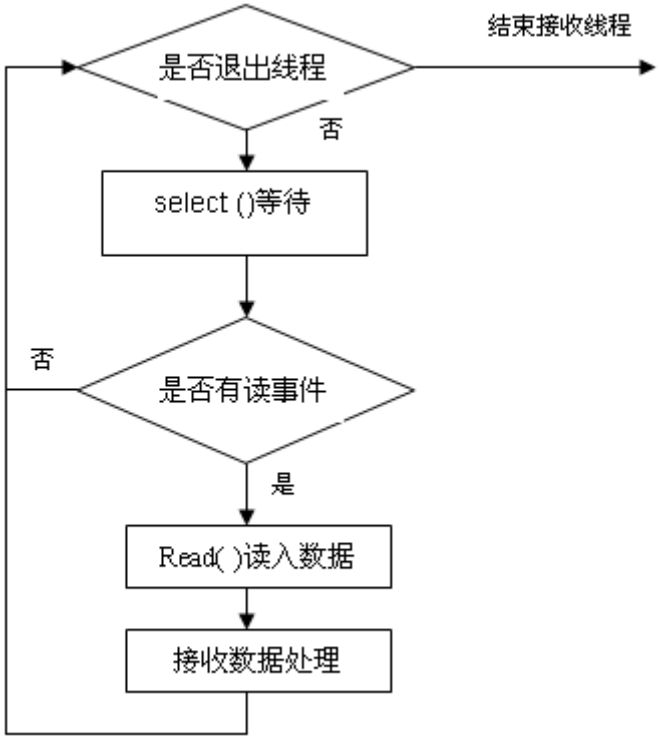
```
// 定义 CAN 通讯类
class EM9X60_CAN
    {
      private:
        // 通讯线程标识符 ID
        pthread_t m_thread;
        // CAN 接收线程
        static int ReceiveThreadFunc( void* lparam );
      public:
        EM9X60_CAN();
        virtual ~EM9X60_CAN();
        // 已打开的 CAN 文件描述符
        int m_fd;
        unsigned int m_canid;

        can_frame rxmsg;

        // 退出数据接收线程标志
        int m_ExitThreadFlag;

        // 按照指定的参数打开 CAN 接口，并创建 CAN 接口接收线程
        int OpenCAN( int CanNo, CAN_BAUDRATE baudrate,
        accept_filter  *AcceptanceFilter );
        // 关闭接口并释放相关资源
        int CloseCAN( );

        // 初始化设置 CAN 数据包 id 信息
        int InitCanIDInfo( struct CanIDInfo* pcanid );
        // CAN 接口写数据
        int WriteCAN( char* Buf, int len );
```

```
    // CAN 接收数据处理函数
    virtual int PackagePro( char* Buf, int len );
};
```

OpenCAN 函数用于根据输入参数打开 CAN 设备，并创建 CAN 数据接收线程。

```
    res = pthread_create( &m_thread, &attr,
                          (void*)&ReceiveThreadFunc,
                          (void*)this );
```

ReceiveThreadFunc 函数是 CAN 数据接收和处理的主要核心代码，在该函数中调用 select( )，等待串口数据的到来。对于接收到的数据处理也是在该函数中实现，在本例程中处理为简单的数据回发，用户可结合实际的应用修改此处代码，修改 PackagePro( )函数即可。流程如下：




```
int EM9X60_CAN::ReceiveThreadFunc(void* lparam)
{
    EM9X60_CAN *pCAN = (EM9X60_CAN*)lparam;
    int len;

    // 定义读事件集合
    fd_set fdRead;
    int ret;
    struct timeval aTime;
```

```
while( 1 )
{
    // 收到退出事件，结束线程
    if( pCAN->m_ExitThreadFlag )
    {
        break;
    }

    FD_ZERO(&fdRead);
    FD_SET(pCAN->m_fd,&fdRead);

    aTime.tv_sec = 0;
    aTime.tv_usec = 30000;

    ret = select( pCAN->m_fd+1,&fdRead,NULL,NULL,&aTime );

    if (ret < 0 )
    {
        pCAN->CloseCAN( );
        break;
    }

    if (ret >= 0)
    {
        // 判断是否读事件
        if (FD_ISSET(pCAN->m_fd,&fdRead))
        {

            len = read(pCAN->m_fd, (char*)&pCAN->rxmsg,
                       sizeof(can_frame) );

            while( len > 0 )
            {
                // 对接收的数据进行处理，这里为简单的数据回发
                pCAN->PackagePro( (char*)&pCAN->rxmsg, len );
                // 处理完毕
                len = read( pCAN->m_fd, (char*)&pCAN->rxmsg,
                sizeof(can_frame) );
            }
        } // 判断是否读事件
    }  // if (ret >= 0)
} // while( 1 )
```

```
    printf( 'ReceiveThreadFunc finished\n' );
    pthread_exit( NULL );
    return 0;
}
```

需要注意的是，select( )函数中的时间参数在 Linux 下，每次都需要重新赋值，否则会自动归 0。