嵌入式 Linux 串口通讯的 C++设计

嵌入式Linux工控板EM9160 提供了 6 个标准异步串口：ttyS1——ttyS6，其中ttyS4、ttyS5、ttyS6 和GPIO的管脚复用，每个串口都有独立的中断模式，使得多个串口能够同时实时进行数据收发。各个串口的驱动均已经包含在嵌入式Linux操作系统的内核中，EM9160在嵌入式Linux系统启动完成时，各个串口已作为字符设备完成了注册加载，用户的应用程序可以以操作文件的方式对串口进行读写，从而实现数据收发的功能。

串口编程接口函数

在嵌入式Linux系统下，所有的设备文件都位于“/dev”目录下，EM9160 上 6 个串口所对应的设备名依次为“/dev/ttyS1”——“/dev/ttyS6”。

嵌入式Linux下操作设备的方式和操作文件的方式是一样的：调用open( )打开设备文件，再调用read( )、write( )对串口进行数据读写操作。这里需要注意的是打开串口除了设置普通的读写之外，还需要设置O_NOCTTY和O_NDLEAY，以避免该串口成为一个控制终端，因为如果作为一个终端有可能会影响到用户的进程。打开的方式如下：

```
sprintf( portname, '/dev/ttyS%d', PortNo ); //PortNo 为串口端口号，从 1 开始
m_fd = open( portname,O_RDWR | O_NOCTTY | O_NONBLOCK);
```

作为串口通讯还需要一些通讯参数的配置，包括波特率、数据位、停止位、校验位等参数。在实际的操作中，主要是通过设置 struct termios 结构体的各个成员值来实现，一般会用到的函数包括：

```
tcgetattr( ) ;
tcflush( );
cfsetispeed( );
cfsetospeed( );
tcsetattr( );
```

其中各个函数的具体使用方法这里就不一一介绍了，用户可以参考嵌入式Linux应用程序开发的相关书籍，也可参看Step2_SerialTest中Serial.cpp模块中set_port( )函数代码。

串口应用的 C++设计

Step2 _SerialTest 是一个支持异步串口数据通讯的示例，该例程采用了面向对象的 C++编程，把串口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串口通讯的操作。

CSerial 类介绍

利用上一小节中介绍的串口 API 函数，封装了一个支持异步读写的串口类 CSerial，CSerial 类中提供了 4 个公共函数、一个串口数据接收线程以及数据接收用到的数据 Buffer。

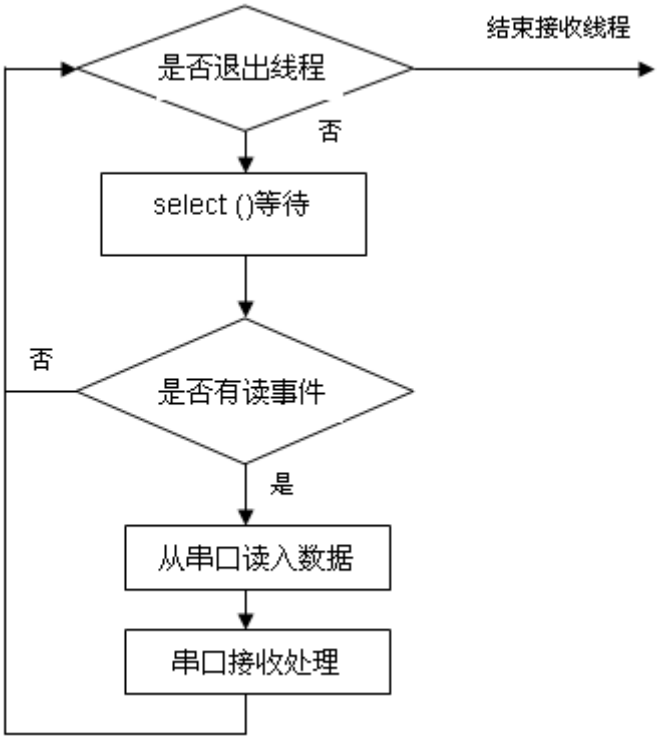
```
class CSerial
{
private:
    //通讯线程标识符 ID
    pthread_t m_thread;
    // 串口数据接收线程
    static int ReceiveThreadFunc( void* lparam );
public:
    CSerial();
    virtual ~CSerial();

    int m_fd; // 已打开的串口文件描述符
    int m_DatLen;
    char DatBuf[1500];
    int m_ExitThreadFlag;

    // 按照指定的串口参数打开串口，并创建串口接收线程
    int OpenPort( int PortNo, int baudrate, char databits, char stopbits, char parity );
    // 关闭串口并释放相关资源
    int ClosePort( );
    // 向串口写数据
    int WritePort( char* Buf, int len );
    // 接收串口数据处理函数
```

```
        virtual int PackagePro( char* Buf, int len );
};
```

OpenPort函数用于根据输入串口参数打开串口，并创建串口数据接收线程。在嵌入式Linux环境中是通过函数pthread_create( )创建线程，通过函数pthread_exit( )退出线程。嵌入式Linux线程属性存在有非分离（缺省）和分离两种，在非分离情况下，当一个线程结束时，它所占用的系统资源并没有被释放，也就是没有真正的终止；只有调用pthread_join( )函数返回时，创建的线程才能释放自己占有的资源。在分离属性下，一个线程结束时立即释放所占用的系统资源。基于这个原因，在我们提供的例程中通过相关函数将数据接收线程的属性设置为分离属性。如：

```
// 设置线程绑定属性
res = pthread_attr_setscope( &attr, PTHREAD_SCOPE_SYSTEM );
// 设置线程分离属性
res += pthread_attr_setdetachstate( &attr, THREAD_CREATE_DETACHED );
```

ReceiveThreadFunc 函数是串口数据接收和处理的主要核心代码，在该函数中调用select( )，阻塞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 CSerial::ReceiveThreadFunc(void* lparam)
{
	CSerial *pSer = (CSerial*)lparam;

	//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( pSer->m_ExitThreadFlag )
		{
			break;
		}
		FD_ZERO(&fdRead);
		FD_SET(pSer->m_fd,&fdRead);
		aTime.tv_sec = 0;
		aTime.tv_usec = 300000;
		ret = select( pSer->m_fd+1,&fdRead,NULL,NULL,&aTime );
		if (ret < 0 )
		{
			//关闭串口
```

```
                pSer->ClosePort( );
                break;
            }
            if (ret > 0)
            {
                //判断是否读事件
                if (FD_ISSET(pSer->m_fd,&fdRead))
                {
                    //data available, so get it!
                    pSer->m_DatLen = read( pSer->m_fd, pSer->DatBuf, 1500 );
                    // 对接收的数据进行处理，这里为简单的数据回发
                    if( pSer->m_DatLen > 0 )
                    {
                        pSer->PackagePro( pSer->DatBuf, pSer->m_DatLen);
                    }
                    // 处理完毕
                }
            }
        }
        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嵌入式Linux中每次都需要重新赋值，否则会自动归 0。

CSerial 类的实现代码请参见 Serial.CPP 文件。

CSerial 类的调用

CSerial 类的具体使用也比较简单，主要是对于类中定义的 4 个公共函数的调用，以下为 Step2_SerialTest.cpp 中相关代码。

```
class CSerial m_Serial;
int main( int argc,char* argv[] )
{
    int i1;
    int portno, baudRate;
    char cmdline[256];

    printf( 'Step2_SerialTest V1.0\n' );
```

```
        // 解析命令行参数：串口号 波特率
        if( argc > 1 ) strcpy( cmdline, argv[1] );
        else portno = 1;
        if( argc > 2 )
        {
                strcat( cmdline, ' ' );
                strcat( cmdline, argv[2] );
                scanf( cmdline, '%d %d', &portno, &baudRate );
        }
        else
        {
                baudRate = 115200;
        }
        printf( 'port:%d baudrate:%d\n', portno, baudRate);
        //打开串口相应地启动了串口数据接收线程
        i1 = m_Serial.OpenPort( portno, baudRate, '8', '1', 'N');
        if( i1<0 )
        {
                printf( 'serial open fail\n');
                return -1;
        }
        //进入主循环，这里每隔 1s 输出一个提示信息
        for( i1=0; i1<10000;i1++)
        {
                sleep(1);
                printf( '%d \n', i1+1);
        }
        m_Serial.ClosePort( );
        return 0;
}
```

从上面的代码可以看出，程序的主循环只需要实现一些管理性的功能，在本例程中仅仅是每隔 1s输出一个提示信息，在实际的应用中，可以把一些定时查询状态的操作、看门狗的喂狗等操作放在主循环中，这样充分利用了嵌入式Linux多任务的编程优势，利用内核的任务调度机制，将各个应用功能模块化，以便于程序的设计和管理。这里顺便再提一下，在进行多个串口编程时，也可以利用本例程中的CSerial类为基类，根据应用需求派生多个CSerial派生类实例，每一个派生类只是重新实现虚函数PackagePro(…)，这样每个串口都具有一个独立的串口数据处理线程，利用Linux内核的任务调度机制以实现多串口通讯功能。