10.1 串口通讯

10.1.1 串口通信介绍

串口是计算机上的串行通信的物理接口。首先先介绍一下串行通信，串行通信的分类：

1、按照数据传送方向，分为：

单工：数据传输只支持数据在一个方向上传输；就像路上的单行线。

半双工：允许数据在两个方向上传输。但是，在某一时刻，只允许数据在一个方向上传输；半双工就像分时段的单行线，上午时段通行这边，下午时段通行另一边，而单工就是全天单行线。

全双工：允许数据同时在两个方向上传输。因此，全双工通信是两个单工通信方式的结合，需要独立的接收端和发送端；全双工就是双向车道。

2、按照通信方式，分为：

同步通信：带时钟同步信号传输。比如：SPI，IIC 通信接口。

异步通信：不带时钟同步信号。比如：UART(通用异步收发器)，单总线。

在同步通讯中，收发设备上方会使用一根信号线传输信号，在时钟信号的驱动下双方进行协调，在异步通讯中不使用时钟信号进行数据同步，通讯中需要双方规约好数据的传输速率（也就是波特率）等，以便更好地同步。常用的波特率有 4800bps、9600bps、115200bps 等。

串口通信中通常使用的是异步串行通信。

串口通信（Serial Communications）的概念非常简单，串口按位（bit）发送和接收字节。尽管比按字节（byte）的并行通信慢，但是串口可以在使用一根线发送数据的同时用另一根线接收数据。它很简单并且能够实现远距离通信。通信使用 3 根线完成，分别是地线（GND）、发送(TXD)、接收(RXD)。由于串口通信是异步的，端口能够在一根线上发送数据同时在另一根线上接收数据。其他线用于握手，但不是必须的。串口通信最重要的参数是波特率、数据位、停止位和奇偶校验。对于两个进行通信的端口，这些参数必须匹配。

10.1.2 串口接头介绍

常用的串口接头有两种，一种是 9 针串口（简称 DB-9），一种是 25 针串口（简称 DB-25）。每种接头都有公头和母头之分，其中带针状的接头是公头，而

带孔状的接头是母头。

以 DB9 为例，如图：

母头：泛指所有带孔状的接头（5 针朝下，从左到右依次是 1~9）。

公头：泛指所有带针状的接头（5 针朝下，从右到左依次是 1~9）。

各引脚的定义：

1 DCD 载波检测
2 RXD 接收数据 ——方向：终端到计算机
3 TXD 发送数据 ——方向：计算机到终端
4 DTR 数据终端准备好
5 GND 信号地线
6 DSR 数据准备好
7 RTS 请求发送
8 CTS 清除发送
9 RI 振铃指示

最简单的三线通行就需要到 2,3,5 接口而已。需要测试串口是否正常时，连接 TXD 接 RXD，RXD 接 TXD，GND 接 GND。自己的 TXD 口接 RXD 口，自发自收，测试串口是否正常。

DB9 和 DB25 的常用信号脚说明：

DB9	DB25
1 数据载波检测 DCD	8 数据载波检测 DCD
2 接收数据 RXD	3 接收数据 RXD
3 发送数据 TXD	2 发送数据 TXD
4 数据终端准备 DTR	20 数据终端准备 DTR
5 信号地 GND	7 信号地 GND
6数据设备准备好DSR	6 数据设备准备好 DSR
7 请求发送 RTS	4 请求发送 RTS
8 清除发送 CTS	5 清除发送 CTS
9 振铃指示 RI	22 振铃指示 RI

GND - Logic Ground

从技术角度讲，GND 不能算是信号。但是没有它其他信号都不能用了。基本上，logic ground 有点像一个参考电压，通过它来判断哪个电压表示正哪个电压表示负。

TXD - Transmitted Data

TXD 信号负载着从你的电脑或者设备到另一端的数据。Mark 范围的电压被解析成 1，而 space 范围电压被解析成 0。

RXD - Received Data

RXD 于 TXD 正好相反。它负载着从另一端的电脑或者设备上传到你的工作站的数据。Mark 和 space 的解析方法于 TXD 一致。

DCD - Data Carrier Detect

DCD 信号通常来自串口连结线的另一端。这条信号线上的 space 电压表示另一端的电脑或者设备现在已经连接。但是，DCD 信号线却不是总可以得到的，有些设备上有这条信号线，而有的则没有。

DTR - Data Terminal Ready

DTR 信号是你的工作站产生的，用以告诉另一端的电脑或者设备你已经是否已经准备好了。Space 电压表示准备好了，而 mark 电压表示没有准备好。当你在工作站上打开串行接口时，DTR 通常自动被设置位有效。

CTS - Clear To Send

CTS 则通常来自连结线的另一端。Space 电压表示你可以从工作站送出更多的数据。CTS 通常用来协调你的工作站和另一端之间的串行数据流。

RTS - Request To Send

如果 RTS 信号被设置成 space 电压，这表示你准备好了一些数据需要传送。和 CTS 一样，RTS 也被用来协调工作站和另一端的电脑或者设备之间的数据流。有些工作站上会一直将这个信号设置位 space。

串口通信还需要注意不同的电平

TTL 电平：TTL 是 Transistor-Transistor Logic，即晶体管-晶体管逻辑的简称，它是计算机处理器控制的设备内部各部分之间通信的标准技术。TTL 电平信号应用广泛，是因为其数据表示采用二进制规定，+5V 等价于逻辑”1”，0V 等价于逻辑”0”。

数字电路中，由 TTL 电子元器件组成电路的电平是个电压范围，规定：

输出高电平>=2.4V，输出低电平<=0.4V；

输入高电平>=2.0V，输入低电平<=0.8V。

RS232 电平：RS232 电平是串口的一个标准。

在 TXD 和 RXD 数据线上：

（1）逻辑 1 为-3~-15V 的电压。

（2）逻辑 0 为 3~15V 的电压。

在 RTS、CTS、DSR、DTR 和 DCD 等控制线上：

（1）信号有效（ON 状态）为 3~15V 的电压。

（2）信号无效（OFF 状态）为-3~-15V 的电压。

这是由通信协议 RS-232 规定的。

RS-232：标准串口，最常用的一种串行通讯接口。有三种类型(A,B 和 C)，它们分别采用不同的电压来表示 on 和 off。最被广泛使用的是 RS-232C，它将 mark(on)比特的电压定义为-3V 到-12V 之间，而将 space(off)的电压定义到+3V 到+12V 之间。传送距离最大为约 15 米，最高速率为 20kb/s。RS-232 是为点对点（即只用一对收、发设备）通讯而设计的，其驱动器负载为 3 ~ 7kΩ。所以 RS-232 适合本地设备之间的通信。

如果用的电平不一致也会导致通信双方通信不上 ,例如 cpu 出来的电平有可能就需要转换才能进行串口通信。

10.1.3 串口编程 API

在本章节中我们使用开源串口库 serial 来进行串口编程的相关应用。

库名称：serial

头文件：<serial/serial.h>

1 安装 serial 库

sudo apt-get install ros-<distro>-serial

其中：

<distro>为你 ros 的版本号，本教程是基于 melodic 版本的 ros 进行，因此该命令具体为：

sudo apt-get install ros-melodic-serial

2 serial 库 API

构造函数

```
serial::Serial::Serial   ( const std::string &       port = "",
                           uint32_t       baudrate = 9600,
                           Timeout        timeout = Timeout(),
                           bytesize_t     bytesize = eightbits,
                           parity_t       parity = parity_none,
                           stopbits_t     stopbits = stopbits_one,
                           lowcontrol_t flowcontrol = flowcontrol_none
)
```

作用：

创建了要给 Serial 对象并且如果端口指定的话打开端口，否则端口保持关闭，直到 serial::Serial::open 被调用。

参数：

port：一个 std::string 包含了串口地址如：/dev/ttyS1、/dev/ttyUSB1；

baudrate：一个表示波特率的无符号 32 位整数如：9600,115200 等；

timeout：一个 serial::Timeout 结构体，定义了串口的超时；

bytesize：每个位组在串口数据传输的大小，默认是 eightbits，参考值有：fivebits, sixbits, sevenbits, eightbits；

parity：奇偶检验的方法，默认是 parity_none，可能的值是：parity_none, parity_odd, parity_even；

stopbits：使用的停止位的数量，默认是 stopbits_one，可能的值是：stopbits_one, stopbits_one_point_five, stopbits_two；

flowcontrol：使用的控制流的类型，默认是 flowcontrol_none，可能的值是：flowcontrol_none, flowcontrol_software, flowcontrol_hardware。

3 串口设置：

void Serial::setPort(const std::string & port)

设置串口端口号：setPort(“/dev/ttyS1”);

void setBaudrate (uint32_t baudrate)

设置波特率：setBaudrate(115200);

void setParity (parity_t parity)

设置串口校验位：setParity(parity_none);
void setStopbits (stopbits_t stopbits)
设置串口通讯停止位：setStopbits(stopbits_one)
void setTimeout (Timeout &timeout)
设置超时时间；
serial::Timeout to = serial::Timeout::simpleTimeout(1000);
ser.setTimeout(to);

4 打开串口：

void open ()

5 关闭串口：

void close ()

6 查询缓存中字符的数量；

size_t available ()

7 写串口缓冲区：

```
size_t  write (const uint8_t *data, size_t size)
```

写入一个指定长度的数组，返回值为实际写入的字节数量；

```
size_t  write (const std::vector< uint8_t > &data)
```

写入一个不指定长度的数组，返回值为实际写入的字节数量；

```
size_t  write (const std::string &data)
```

写入一个字符串，返回值为实际写入的字节数量；

8 读取串口缓冲区：

```
size_t  read (uint8_t *buffer, size_t size)
```

读取指定长度的内容到 `buffer` 中，返回值为实际读到的字节数量；

```
size_t  read (std::vector< uint8_t > &buffer, size_t size=1)
size_t  read (std::string &buffer, size_t size=1)
std::string  read (size_t size=1)
size_t  readline (std::string &buffer, size_t size=65536,
std::string eol="\n")
std::string  readline (size_t size=65536, std::string eol="\n")
std::vector< std::string >  readlines (size_t size=65536,
std::string eol="\n")
```

9 清理缓冲区：

void flush ()
清空读写缓冲区；
void flushInput ()

清空读缓冲区；

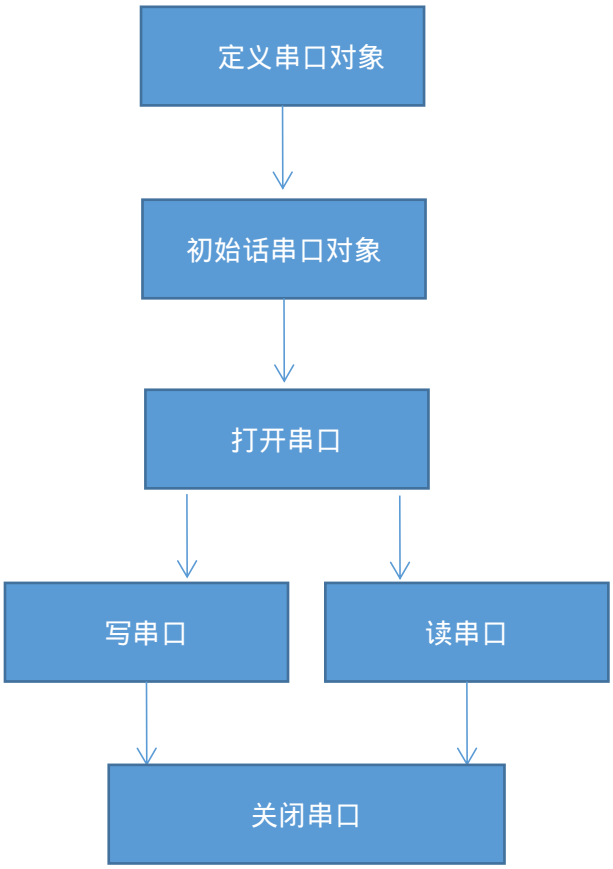
void flushOutput ()

清空写缓冲区

更多 API 请参考以下网站：

http://docs.ros.org/en/kinetic/api/serial/html/classserial_1_1Serial.html#afa2c1f9114a37b7d140fc2292d1499b9

10.1.4 串口编程流程




```
#include <ros/ros.h>
#include <serial/serial.h>
#include <iostream>
int main(int argc, char** argv)
{
    ros::init(argc, argv, "serial_port");
    //创建句柄（虽然后面没用到这个句柄，但如果不创建，运行时进程会出错）
    ros::NodeHandle n;
```

```
//创建一个 serial 类
serial::Serial sp;
//创建 timeout
serial::Timeout to = serial::Timeout::simpleTimeout(100);
//设置要打开的串口名称
sp.setPort("/dev/ttyUSB0");
//设置串口通信的波特率
sp.setBaudrate(115200);
//串口设置 timeout
sp.setTimeout(to);

try
{
    //打开串口
    sp.open();
}
catch(serial::IOException& e)
{
    ROS_ERROR_STREAM("Unable to open port.");
    return -1;
}
//判断串口是否打开成功
if(sp.isOpen())
{
    ROS_INFO_STREAM("/dev/ttyUSB0 is opened.");
}
else
{
    return -1;
}
ros::Rate loop_rate(500);
while(ros::ok())
{
    //获取缓冲区内的字节数
    size_t n = sp.available();
```

```
        if(n!=0)
        {
            uint8_t buffer[1024];
            //读出数据
            n = sp.read(buffer, n);

            for(int i=0; i<n; i++)
            {
                //16 进制的方式打印到屏幕
                std::cout << std::hex << (buffer[i] & 0xff) << " ";
            }
            std::cout << std::endl;
            //把数据发送回去
            sp.write(buffer, n);
        }
        loop_rate.sleep();
    }
    //关闭串口
    sp.close();
    return 0;
}
```

10.2 modbus 通讯协议

10.2.1 modbus 协议简介

Modbus 通信协议是最早的到目前为止最流行的自动化协议在过程自动化和 SCADA 领域（监督控制和数据采集）。了解如何创建基于 Modbus 的网络对任何电气工程师都是十分重要和必要的。

Modbus 是一种通信协议由 Modicon 于 1979 年出版用于其可编程逻辑控制器（PLC）。

Modicon 现在由施耐德电气拥有。Modbus 提供通用语言用于彼此通信的设备和设备。

例如，Modbus 启用系统上的设备测量连接在同一网络上的温度和湿度将结果传达给监控计算机或 PLC。以及 Modbus 协议的开发和更新由 Modbus 组织管理。Modbus 组织是用户和供应商的协会符合 Modbus 标准的设备。

串行端口（RS-485）存在多个版本的 Modbus 协议，而以太网最常见的是 ModbusRTU，ModbusASCII，ModbusTCP 和 ModbusPlus。

Modicon 发布了 Modbus 通信接口用于基于主/从架构的多点网络。实现 Modbus 节点之间的通信带有发送请求和读取响应类型的消息。Modbus 是一个描述的开放标准消息传递通信对话框。

Modbus 通过多种类型的物理介质进行通信例如串行 RS-232，RS-485，RS-422 和以太网。选择设备时请确认需要的物理介质。

起始位	设备地址	功能代码	数据	CRC 校验	结束符
T1-T2-T3-T4	8Bit	8Bit	n 个 8Bit	16Bit	T1-T2-T3-T4

10.2.2 libmodbus 库编程应用

libmodbus 库编程应用基本分为以下几个步骤：

1、初始化 RTU 指针

modbus_t *modbus_new_rtu(const char *device, int baud, char parity, int data_bit, int stop_bit)

2、2、设置从站 ID

int modbus_set_slave(modbus_t *ctx, int slave);

3、建立连接

int modbus_connect(modbus_t *ctx);

4、读取保持寄存器/读取输入寄存器/读取位

读取位（读取线圈状态）：

int modbus_read_bits(modbus_t *ctx, int addr, int nb, uint8_t *dest);

读取输入位（读取输入状态）：

int modbus_read_input_bits(modbus_t *ctx, int addr, int nb, uint8_t *dest);

读取保持寄存器 ：

int modbus_read_registers(modbus_t *ctx, int addr, int nb, uint16_t *dest);

读取输入寄存器：

int modbus_read_input_registers(modbus_t *ctx, int addr, int nb, uint16_t *dest);

读取控制器（controller）描述：

int modbus_report_slave_id(modbus_t *ctx, int max_dest, uint8_t *dest);

5、写单寄存器/写多寄存器/写多位数据

写一位数据（强置单线圈）：

int modbus_write_bit(modbus_t *ctx, int addr, int status);

写单寄存器（预置单寄存器）：

int modbus_write_register(modbus_t *ctx, int addr, int value);

写多位数据（强置多线圈）：

int modbus_write_bits(modbus_t *ctx, int addr, int nb, const uint8_t *src);

写多寄存器（预置多寄存器）：

int modbus_write_registers(modbus_t *ctx, int addr, int nb, const uint16_t *src);

6、关闭连接

void modbus_close(modbus_t *ctx);

```
#include "ros/ros.h"
//modbus 库头文件
#include "modbus/modbus-rtu.h"
//向量头文件
#include "vector"
#include "string"

//速度寄存器地址
#define READ_SPEED 0
//电压寄存器地址
#define READ_POWER_VOLTAGE   16
//陀螺仪寄存器地址
#define READ_MPU   17


using namespace std;

uint16_t mbbuf[35];
uint16_t probeen=0;
//实例化底盘消息
modbus_t *mb=NULL;

void translate_mb_data(void)
{
    //共用体
    union {
        double dd;
        uint16_t id[4];
    } d_to_uint;
    //各个轮子转速
    uint16_t *mbp=&mbbuf[READ_SPEED];
    for(int i=0; i<4; i++) {
```

```
            d_to_uint.id[3]=*mbp++;
            d_to_uint.id[2]=*mbp++;
            d_to_uint.id[1]=*mbp++;
            d_to_uint.id[0]=*mbp++;
            double speed =d_to_uint.dd;
        }

}
//car_cmd 消息回调函数
void twistCallback(const fox_msgs::car_cmd & mycmd)
{
    if(mycmd.speed.size()!=4) return;
    union {
        double dd;
        uint16_t id[4];
    } d_to_uint;
    uint16_t send_data[16];
    uint16_t *p=send_data;
    for(int i=0; i<4; i++) {
        d_to_uint.dd=mycmd.speed[i];
        *p++=d_to_uint.id[3];
        *p++=d_to_uint.id[2];
        *p++=d_to_uint.id[1];
        *p++=d_to_uint.id[0];
    }
    //写入寄存器
    int res = modbus_write_registers(mb,0,16,send_data);
}
int main(int argc, char **argv)
{


            if(!(mb=modbus_new_rtu(curr_port.c_str(),115200,'N',8,
            1))) return ;
            //设置从机地址为 1 如果返回-1 则为失败
            if(modbus_set_slave(mb,1)==-1) {
                modbus_free(mb);
                return ;
            }
            if(modbus_connect(mb)==-1) {
                modbus_free(mb);
                return;
            }
            modbus_set_response_timeout(mb,0,200000);
        //设置速度消息长度 4,imu 消息长度 9
        //初始化节点
    ros::init(argc, argv, "car_controller");
```

```
    ros::NodeHandle nh;
    //创建发布者 car_data
    ros::Rate loop_rate(20);
    while (ros::ok()) {
        //动态内存分配
        memset(mbbuf,0,35*2);
        usleep(2*1000);
        //请求 modbus
        if(modbus_read_input_registers(mb,0,35,mbbuf)==35) {
            translate_mb_data();
        }
        ros::spinOnce();
        loop_rate.sleep();
    }
    modbus_close(mb);
    modbus_free(mb);
    return 0;
}
```

libmodbus 编译安装：CMakeLists.txt 中添加

```
include (ExternalProject)

ExternalProject_Add(modbus-build

  INSTALL_DIR ${CMAKE_CURRENT_SOURCE_DIR}/modbus

  URL ${CMAKE_CURRENT_SOURCE_DIR}/libmodbus-3.1.4.tar.gz

  CONFIGURE_COMMAND                    <SOURCE_DIR>/configure

--prefix=<INSTALL_DIR>

  STEP_TARGETS install

)

ExternalProject_Get_Property(modbus-build install_dir)
```

libmodbus-3.1.4.⋯